Vom Fachbereich für Mathematik und Informatik

der Technischen Universität Braunschweig

genehmigte Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften (Dr.rer.nat.)

Thomas Gehrke

Dynamische Modelle für Reaktive Systeme mit Daten

11. Dezember 2000

1. Referentin: Prof. Dr. Ursula Goltz

2. Referent: Prof. Dr. Hans-Dieter Ehrich

eingereicht am: 10. August 2000

Für Tatjana

Danksagung

Diese Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter an der Universität Hildesheim und der Technischen Universität Braunschweig.

Meiner Mentorin Prof. Ursula Goltz danke ich für die Ermöglichung meiner Arbeit als wissenschaftlicher Mitarbeiter und für den nötigen wissenschaftlichen Freiraum. Herrn Prof. Ehrich danke ich für die Übernahme des zweiten Gutachtens. Meinen ehemaligen Hildesheimer Kollegen Peter Niebert und Heike Wehrheim sowie meinen Braunschweiger Kollegen Karsten Diethers, Thomas Firley, Jürgen Spieß, Werner Struckmann und Bernhard Witte danke ich für die gemeinsame Arbeit und die kollegiale Atmosphäre.

Meinen besonderer Dank gilt Michaela Huhn und Arend Rensink, die meine wissenschaftliche Arbeit wesentlich geprägt haben. Michaela Huhn danke ich für die zahlreichen Diskussionen und die anfänglichen Ratschläge in bezug auf wissenschaftliche Arbeitsweise. Arend Rensink verdanke ich mein Wissen über die Beschreibung paralleler Systeme mit Prozeßalgebren. Er hat den ersten Teil dieser Arbeit durch Vorschläge, Kommentare und Diskussionen begleitet.

Meinen Eltern danke ich für die Unterstützung und die Ermöglichung meines Hochschulstudiums. Ohne sie wäre diese Arbeit niemals möglich gewesen.

Meiner Frau Tatjana danke ich für die Geduld, die sie mit mir während der Endphasen der Arbeit gehabt hat und für die bisherige schöne Zeit.

Braunschweig, im Dezember 2000 Thomas Gehrke

Zusammenfassung

In dieser Arbeit erweitern wir Formalismen für reaktive Systeme um die funktionale Beschreibung von Daten, um eine integrierte Spezifikation von Verhaltens- und Datenspezifikation reaktiver Systeme zu ermöglichen. Dabei legen wir nicht die Verwendung einer bestimmten Datensprache fest, sondern definieren Schnittstellen, über die verschiedene funktionale Sprachen integriert werden können. Zusätzlich unterstützen die Formalismen die Modellierung von Systemen mit dynamischer Struktur, bei denen zur Laufzeit neue Kommunikationsverbindungen erzeugt und bestehende Verbindungen verändert werden können.

Die Bedeutung der Sprachkonstrukte wird durch formale Semantiken beschrieben. Hierbei wird die Semantik der Verhaltenssprache mit der Semantik der jeweils gewählten Datensprache parametrisiert, so daß eine integrierte formale Semantik entsteht. Zusätzlich werden Äquivalenzrelationen definiert, um die Analyse von Spezifikationen durch Äquivalenzbetrachtungen zu ermöglichen.

Im ersten Teil der Arbeit definieren wir eine Prozeßalgebra, die zum einen durch das aus dem π-Kalkül bekannte Konzept der *Mobilität* die Beschreibung dynamischer Systemstrukturen unterstützt, und zum anderen die Integration funktionaler Datensprachen erlaubt. Der Operator zur Beschreibung nebenläufigen Verhaltens orientiert sich an den Konstrukten zur Prozeßerzeugung aus parallelen Programmiersprachen. Die Bedeutung der Operatoren der Algebra wird durch eine operationelle Semantik festgelegt. Weiterhin werden als Äquivalenzrelationen auf Prozessen die starke und die schwache *Bisimulation* definiert und durch axiomatische Semantiken charakterisiert. Auf diese Weise wird die Analyse von Spezifikationen durch Äquivalenzbetrachtungen ermöglicht. Als Beispiel für die Integration einer Datensprache verwenden wir einen erweiterten, getypten λ-Kalkül. Anhand einer Fallstudie wird die Einsetzbarkeit der axiomatischen Semantik für die Systemanalyse demonstriert.

Im zweiten Teil der Arbeit führen wir eine Variante von Interaktionsdiagrammen ein, die speziell zur Spezikation verteilter Systeme mit Daten entworfen wurde. Die Beschreibung komplexen Systemverhaltens wird durch Operatoren wie z.B. Alternativen und Schleifen unterstützt. Weiterhin ist es möglich, die Beschreibung eines Systemverhaltens durch Komposition einzelner Diagramme anzugeben. Aufgrund der verteilten Systemstruktur wird in diesen Konstrukten jeweils eine Systemkomponente benannt, die die Ausführung des Konstrukts kontrolliert. Die formale Semantik der Diagramme wird, ausgehend von einer textuellen Darstellung der Diagramme, durch eine Übersetzung in Terme der im ersten Teil der Arbeit vorgestellten Prozeßalgebra definiert, da diese bereits eine formale Notation für eine integrierte Verhaltens- und Datenspezifikation realisiert. Außerdem können Analysen von Diagrammen durch Anwendung der axiomatischen Semantik der Prozeßalgebra durchgeführt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Moderne Softwaresysteme zeichnen sich zunehmend durch eine wachsende Komplexität aus. Diese Systeme steuern technische Prozesse, agieren in Netzwerken wie z.B. dem Internet oder verwalten große Datenbestände. Viele Softwaresysteme sind verteilte Systeme, in denen Komponenten nebenläufig arbeiten und miteinander durch Kommunikation Daten austauschen. Insbesondere gewinnen *reaktive Systeme*, die in einem permanenten Datenaustausch mit ihrer Umwelt stehen, an Bedeutung.

Durch den wachsenden Komplexitätsgrad und die gestiegenen Anforderungen an Software ist daher neben der Implementierung die *Modellierung* von Softwaresystemen ein zentraler Aspekt der Softwareentwicklung. Bei der Modellierung werden zum einen die Anforderungen an die Software spezifiziert, zum anderen wird das System durch eine Abfolge von Designphasen durch die Erstellung und Verfeinerung von Spezifikationen schrittweise entworfen. Hierbei ist insbesondere der Einsatz *formaler* Notationen, deren Sprachkonstrukte durch eine formale Semantik definiert sind, von Bedeutung. Die Existenz einer formalen Semantik erlaubt zum einen eine direkte Umsetzung der Spezifikation in Programmcode, zum anderen sind für viele Notationen spezielle Techniken entwickelt worden, die die Analyse und Verifikation von Systemeigenschaften bereits vor der Implementierungsphase unterstützen. So können durch den Einsatz von Analysetechniken Designfehler bereits vor der Implementierung und dem darauf folgenden Testen gefunden werden.

Die zu spezifizierenden Aspekte von Softwaresystemen lassen sich einteilen in *Struktur*, *Verhalten* und *Daten*. Viele Notationen zur Spezifikation von Systemeigenschaften unterstützen entweder nur die Struktur- und Verhaltensspezifikation und abstrahieren von den Datentransformationen, oder sie beschreiben die Daten und erlauben keine Angaben zu Struktur und Verhalten von Systemen. Werden verschiedene Formalismen zur Modellierung von Daten- und Verhaltensaspekten innerhalb eines Entwurfs kombiniert, stellt sich die Frage nach der Integration der beiden formalen Semantiken, um eine integrierte Analyse beider Systemaspekte durchführen zu können. Daher ist die Entwicklung von formalen Notationen,

die sowohl die Daten- als auch die Verhaltensaspekte von Systemen modellieren können, wünschenswert.

Oft ist in nebenläufigen Systemen die Systemstruktur *dynamisch*: es werden neue Kommunikationsverbindungen erzeugt und existierende Verbindungen an neue Anforderungen angepaßt. Ein Beispiel hierfür sind die mobilen Telefone, bei denen der Anwender je nach Aufenthaltsort über verschiedene Verbindungsstationen in das Mobilnetz eingebunden wird. Die Eigenschaft der dynamischen Kommunikationsverbindungen bezeichnen wir als *Mobilität*. Zur Spezifikation mobiler Systeme muß die Notation das Konzept der dynamischen Verbindungen durch Syntax und Semantik unterstützen.

In dieser Arbeit werden zwei Sprachen zur Spezifikation reaktiver Systeme vorgestellt, die eine integrierte Beschreibung des Systemverhaltens und der damit verbundenen Datentransformationen ermöglichen. Die erste Sprache ist textorientiert und basiert auf der Theorie der Prozeßalgebren. Die zweite Sprache ist graphischer Natur; sie verbindet die Sprachelemente aus Interaktionsdiagrammen zur Darstellung verteilten Verhaltens mit Konzepten zur Behandlung von Daten. Beide Sprachen erhalten eine formale Semantik, die die Bedeutung der jeweiligen Sprachelemente definiert. Durch ihre formale Verbindung von Daten- und Verhaltensspezifikation ist die Textsprache als semantische Grundlage für andere Sprachen geeignet. So wird die Semantik der graphischen Sprache durch eine Übersetzung der Diagrammelemente in Terme der Textsprache definiert.

1.1 Vorbemerkungen

In diesem Abschnitt ordnen wir den Inhalt der vorliegenden Arbeit in die relevanten Themenkomplexe ein.

1.1.1 Reaktive Systeme

Der Begriff der *reaktiven Systeme* wurde von Pnueli eingeführt [Pnu86, MP92]. Er bezeichnet Systeme, die im Gegensatz zu den herkömmlichen *transformationellen* Programmsystemen fortwährend mit ihrer Umgebung interagieren.

Transformationelle Systeme. Transformationelle Systeme überführen eine Menge von Eingaben in eine Menge von Ausgaben. Zu Beginn des Systemablaufs erhalten sie Eingabedaten, die vom System verarbeitet werden. Liegen Ergebnisse der Datenverarbeitung vor, werden diese an den Aufrufer zurückgegeben und das System beendet seine Ausführung. Beispiele für diese Art von Systemen sind Übersetzer von Programmiersprachen sowie Programme zum Lösen von Gleichungssystemen. Transformationelle Systeme besitzen nur eine begrenzte Laufzeit und interagieren während dieser nur in geringem Umfang mit ihrer Umgebung.

Reaktive Systeme. Reaktive Systeme werden durch ihre ständige Interaktion mit ihrer Umgebung charakterisiert. Ein reaktives System kommuniziert während seiner Ausführung fortlaufend mit seiner Umgebung, so daß ein permanenter Datenaustausch stattfindet. Das System reagiert auf Ereignisse aus seiner Umgebung, daher wird das Systemverhalten in hohem Maße durch die Umgebung beeinflußt.

Reaktive Systeme bestehen meist aus Teilsystemen, die wiederum aus lokalen Teilsystemen bestehen können. Diese Teilsysteme sind i. allg. selbst reaktive Systeme, die durch Kommunikation interagieren. Es ergibt sich somit eine hierarchische Gliederung des Systems. Das Gesamtsystem wird durch die Komposition seiner Teilsysteme und die Interaktion zwischen den Komponenten gebildet. Reaktive Systeme weisen meist einen hohen Grad an Nebenläufigkeit auf, da zum einen die Teilsysteme parallel ausgeführt werden und zum anderen das Gesamtsystem parallel zu seiner Umgebung abläuft.

Typische Einsatzgebiete für reaktive Systeme sind die Steuerung technischer Vorgänge (z.B. in Fahrzeugen), Betriebssysteme und graphische Oberflächen. Die in diesen Bereichen auftretenden Anforderungen an Kommunikation und Reaktion des Systems auf wechselnde Bedingungen in seiner Umgebung lassen sich durch herkömmliche Methoden der transformationellen Programmierung nicht erfüllen.

Der komplexe Aufbau reaktiver Systeme und der damit verbundene hohe Grad an Kommunikation zwischen den Systemkomponenten untereinander und mit der Systemumgebung impliziert einen erhöhten Aufwand bei der Spezifikation und der Implementierung solcher Systeme. Daher wurden spezielle Formalismen zur Spezifikation reaktiver Systeme entwickelt, die die Beschreibung der oben genannten Eigenschaften dieser Systeme ermöglichen.

1.1.2 Spezifikation

Formalismen zur Spezifikation reaktiver Systeme müssen in der Lage sein, die Beschreibung der typischen Eigenschaften und Konzepte dieser Systeme, wie z.B. Parallelität und Interaktion, zu ermöglichen. Daher wurden spezielle Formalismen entwickelt, die die Darstellung dieser Konzepte unterstützen. Hierzu gehören z.B. *Prozeßalgebren* [BW90, Fok00, Hoa85, Mil89], *Petri-Netze* [Rei90, JR91], *Ereignisstrukturen* [Win87, LG91] sowie graphische Spezifikationssprachen wie die *Message Sequence Charts* [ITU96a, ITU99], *Statecharts* [Har87, HP98] und die dynamischen Modelle der *Unified Modeling Language* (*UML*) [BRJ98, OMG99]. Für die vorliegende Arbeit sind insbesondere die Prozeßalgebren und die erwähnten graphischen Notationen relevant, daher geben wir im folgenden eine kurze Übersicht über diese Spezifikationstechniken.

Prozeßalgebren

Prozeßalgebren sind formale Sprachen, in denen reaktive Systeme und ihr Verhalten spezifiziert werden können. Dabei konzentrieren sich Prozeßalgebren auf die Verhaltensaspekte von Systemen und abstrahieren i. allg. von den Datentransformationen.

Das Verhalten von Systemen wird in Form von *Prozessen* angegeben. Ein Prozeß ist eine Folge von *Aktionen*: Aktionen sind die elementaren und unteilbaren Grundeinheiten der Ausführung von Prozessen. Beispiele für Aktionen sind das Senden und Empfangen von Werten durch Kommunikation. Prozesse werden durch die Verknüpfung von Aktionen durch die *Operatoren* der jeweiligen Prozeßalgebra gebildet. Im Gegensatz zu den meisten Programmiersprachen werden bei der Termbildung algebraische Gesetze eingehalten und es sind Umformungen zwischen äquivalenten Termen möglich (siehe unten). Zu den Operatoren gehören z.B. die sequentielle und die parallele Komposition von Prozessen sowie die Auswahl zwischen mehreren Alternativen. Die Kommunikation zwischen Prozessen wird durch Kommunikationskanäle realisiert, über die Prozesse Nachrichten senden und empfangen können. Wichtige Prozeßalgebren sind *ACP* (*Algebra of Communicating Processes*) von Baeten und Weijland [BW90], *CCS* (*Calculus of Communicating Systems*) von Milner [Mil89] und *CSP* (*Communicating Sequential Processes*) von Hoare [Hoa85].

Mobilität. Den genannten Prozeßalgebren ist gemeinsam, daß die Menge von Kommunikationskanälen innerhalb der Ausführung des beschriebenen Systemverhaltens statisch ist. Es ist weder möglich, während der Ausführung eines Systems neue Kanäle zu erzeugen noch Kanäle als Parameter von Nachrichten an andere Prozesse zu übertragen. Daher wurden Prozeßalgebren entwickelt, die um das Konzept der Mobilität erweitert wurden: Diese Algebren erlauben sowohl das dynamische Erzeugen neuer Kanäle als auch das Übertragen von Kanälen in Kommunikationsaktionen. Auf diese Weise lassen sich Systeme mit dynamischen Kommunikationsstrukturen spezifizieren. Die bekannteste Algebra mit Mobilität ist der *π-Kalkül* [MPW92] von Milner, Parrow und Walker, der eine Erweiterung von CCS um das Konzept der Mobilität darstellt. Weitere Ansätze sind der *Fusion Calculus* [PV98] von Parrow und Victor und die *Mobile Ambients* [CG97] von Cardelli und Gordon. Der letzte Kalkül intergriert darüber hinaus ein Lokalitätenkonzept, so daß die Zuordnung von Prozessen an ausführende Komponenten beschreibbar wird.

Semantik. Die Bedeutung der Aktionen und Operatoren einer Prozeßalgebra wird durch die Angabe einer formalen *Semantik* definiert. Hierbei unterscheiden wir drei Arten der Semantik:

- Die *operationelle Semantik* bildet die Ausführung von Prozessen auf Modelle ab. In den Modellen werden die elementaren Schritte bei der Ausführung

durch Übergänge zwischen den Zuständen des Systems dargestellt. Verbreitete Modellklassen sind Transitionssysteme [Plo81], Ereignisstrukturen [Win87] sowie Petri-Netze [Rei90]. Bei der Verwendung von Transitionssystemen wird die parallele Ausführung unabhängiger Aktionen durch eine zeitlich verschachtelte Ausführung dieser Aktionen modelliert (*Interleaving*-Semantik). Semantiken auf der Basis von Ereignisstrukturen oder Petri-Netzen bilden hingegen die parallele Ausführung von Prozessen auf nebenläufige Teilmodelle ab (siehe z.B. [Gol88]).

- In der *denotationellen Semantik* werden die Operatoren der Algebra auf Operatoren auf den entsprechenden Modellen abgebildet. Eine mögliche Modellklasse sind Transitionssysteme in Form von Bäumen; in diesem Fall werden die Kompositionsoperatoren der Algebra auf Kompositionsoperatoren von Bäumen abgebildet.

- Bei der *axiomatischen Semantik* werden Äquivalenzklassen von Prozessen gebildet, die bzgl. eines Beobachters gleiches Verhalten aufweisen. Diese Äquivalenzklassen werden durch Gleichungen beschrieben, mit deren Hilfe äquivalente Prozesse ineinander überführt werden können. Auf diese Weise lassen sich Prozesse mit komplexen Operatoren in einfachere, äquivalente Prozesse umformen. Von zentraler Bedeutung ist hierbei die Äquivalenzrelation, bzgl. der die Prozesse einer Klasse gleich gesetzt werden. Verschiedene Äquivalenzbegriffe werden z.B. in [vG90, vG93] diskutiert.

Für Prozeßalgebren werden meist operationelle Semantiken definiert, die die Ausführung von Prozessen beschreiben. Diese Semantiken erzeugen oft zu detailreiche Modelle, so daß von diesen durch die Definition von Verhaltensäquivalenzen abstrahiert wird. Außerdem können die Äquivalenzrelationen zur Analyse von Prozessen verwendet werden. Wohluntersuchte Verhaltensäquivalenzen sind die *starke* und die *schwache Bisimulation* [Mil89]. Diese Relationen betrachten zwei Prozesse als äquivalent, wenn sie sich wechselseitig in ihren Zuständen simulieren können. Hierbei wird bei der starken Bisimulation auch das interne Verhalten von Prozessen betrachtet, während die schwache Bisimulation von internem Verhalten abstrahiert. Äquivalenzrelationen werden oft durch die Angabe einer axiomatischen Semantik charakterisiert. Hierbei werden die *Korrektheit* und die *Vollständigkeit* des Gleichungssystems bzgl. der entsprechenden Äquivalenzrelation nachgewiesen. Ein Gleichungssystem ist korrekt, wenn zwei durch das System als gleich betrachtete Terme äquivalente Semantiken besitzen. Ein Gleichungssystem ist vollständig, wenn zwei Terme mit äquivalenten Semantiken auch durch das System als gleich identifiziert werden. Die Eigenschaft der Korrektheit ist für die praktische Anwendbarkeit des Gleichungssystems von zentraler Bedeutung, da diese Eigenschaft gewährleistet, daß Termumformungen nur zwischen semantisch äquivalenten Termen vorgenommen werden. Die Eigenschaft der Vollständigkeit

ist hingegen eher von theoretischem Interesse; sie besagt, daß keine weiteren Axiome zur Charakterisierung der entsprechenden Äquivalenz notwendig sind.

Graphische Notationen

Reaktive Systeme bestehen meist aus nebenläufigen Komponenten, die durch den Austausch von Nachrichten interagieren. Daher weisen nebenläufige Systeme oft komplexe Abläufe auf, die aus einer textuellen Darstellung nicht ohne weiteres direkt ersichtlich sind. Daher werden verstärkt graphische Spezifikationsmethoden verwendet, um die Systeme und ihre möglichen Ausführungen zu visualisieren. Beispiele für solche Notationen sind die *Petri-Netze* [Rei90], *SDL* [Hog89, OFMP+94] und *Estelle* [Hog89, TG97].

In letzter Zeit hat sich die *Unified Modeling Language* (UML) von Booch, Rumbaugh und Jacobson [BRJ98] als Quasi-Standard für die graphische Spezifikation (objektorientierter) Systeme etabliert. UML besteht aus einer Menge von verschiedenen Diagrammtypen, die zur Modellierung der unterschiedlichen Aspekte von Systemen verwendet werden können. Neben Diagrammtypen wie den Klassendiagrammen zur Beschreibung der Daten enthält UML auch diverse Diagramme zur Beschreibung des Systemverhaltens. Obwohl UML in erster Linie für die Modellierung objektorientierter Systeme gedacht ist, läßt sich auch das Verhalten anderer Systeme mit Hilfe der Verhaltensdiagramme spezifizieren.

Die Verhaltensdiagramme von UML lassen sich grob in zwei Klassen einteilen. Die erste Klasse beschreibt das interne Verhalten von Systemkomponenten. Diese Notationen sind meist zustandsbasiert und beschreiben das Verhalten der Komponente durch Zustandsübergänge, die durch interne oder externe Ereignisse ausgelöst werden. Zu diesem Zweck beinhaltet UML die *Statecharts* von Harel [Har87, HP98]. Die zweite Klasse von Diagrammtypen bilden die *Interaktionsdiagramme*, die zur Beschreibung der Interaktion zwischen den Komponenten von Systemen verwendet werden. Diese Diagramme stellen die möglichen Abläufe eines Systems durch Sequenzen von Nachrichten dar. Zu dieser Klasse gehören die *Sequenz-* und *Kollaborationsdiagramme* (siehe auch [JCJÖ92]).

Mit den Interaktionsdiagrammen eng verwandt sind die *Message Sequence Charts* (MSC) [ITU96a, ITU99, Ren99], die ebenfalls den Nachrichtenfluß zwischen den Systemkomponenten bzw. zwischen dem System und seiner Umgebung beschreiben. Über die Ausdrucksfähigkeit von UML hinaus unterstützen die Message Sequence Charts zusätzlich eine hierarchische Systembeschreibung durch die Spracherweiterung *High-level MSC* [MR97, Ren99]. In diesem Ansatz können Diagramme als Knoten in übergeordneten Diagrammen verwendet werden, so daß sich die Beschreibung des gesamten Systemverhaltens durch die hierarchische Komposition von Diagrammen angeben läßt.

Graphische Notationen sind i. allg. halbformal. Die Syntax der Diagramme ist durch die Definition der graphischen Elemente festgelegt. Bei der Semantik ist zu unterscheiden zwischen der Festlegung der Beziehungen zwischen

den graphischen Elementen der Diagramme und der Bestimmung der möglichen Ausführungen des beschriebenen Systems. Für den ersten Fall verwendet UML ein Metamodell, das die Relationen zwischen Diagrammelementen in Form von UML-Diagrammen definiert. Zur Angabe der Verhaltenssemantik ist eine zusätzliche Übersetzung der Diagramme in eine formale Notation zur Verhaltensbeschreibung notwendig. So existieren Übersetzungen von Interaktionsdiagrammen in Prozeßalgebren [MR94a, ITU96b, MR97, GHRW98], in Petri-Netze [GRG93, GGW98, GGW99, Stö99] und in temporallogische Formeln [Kna99]. Die Definition dieser Semantiken hat ergeben, daß einige der in den graphischen Notationen enthaltenen Konzepte eine mehrdeutige Semantik haben. Dies gilt insbesondere bei der Spezifikation verteilter Systeme; hier ist z.B. die exakte Bedeutung von bedingten Nachrichten und Kompositionsoperatoren mit Alternativen problematisch [BM95, GGW98, GGW99].

1.1.3 Daten

Neben dem prozeßorientierten Verhalten eines Systems, das durch die Synchronisation und Kommunikation der Teilsysteme untereinander und mit der Systemumgebung bestimmt wird, ist die Behandlung der vom System verwendeten Daten von großer Bedeutung. Hierbei ist wichtig, daß Daten und Datentransformationen in einer problemorientierten Weise spezifiziert werden können, damit die Spezifikation nicht durch unzureichende Beschreibungsmittel verkompliziert wird. Ebenso ist eine integrierte Semantik für sowohl Verhaltens- als auch Datenbeschreibung von Bedeutung, damit zum einen die Anwendung von Analyse- und Verifikationsverfahren ermöglicht und zum anderen die Umsetzung der Spezifikationen in ablauffähigen Programmcode unterstützt wird.

Daten in Prozeßalgebren

Die im vorherigen Abschnitt vorgestellten Spezifikationsmethoden stellen die Struktur und das Verhalten von reaktiven Systemen dar, abstrahieren aber i. allg. von den Datentransformationen, die während des Ablaufs in dem reaktiven System durchgeführt werden.

In der Basistheorie der Prozeßalgebren ist die Behandlung von Daten nicht vorgesehen, da sich diese Formalismen auf das Systemverhalten konzentrieren. Daher wurden für Prozeßalgebren Erweiterungen definiert, die die Behandlung von Daten in die Konstrukte der Prozeßalgebra integrieren. Hierzu gehören die von Hennessy und Ingolfsdottir in [HI93] vorgestellte Algebra, die von der ISO standardisierte Spezifikationssprache *LOTOS* [BB87, ISO87, Hog89] mit der algebraischen Datenbeschreibungssprache *ACT ONE*, die Sprache *Pict* von Pierce und Turner [PT95, PT97] sowie die Sprache *ProFun* [Geh96, GH96] des Autors, die eine Prozeßsprache mit einer funktionalen Datensprache verbindet.

In diesen Ansätzen ist die Datensprache vorgegeben, so daß eine problemorientierte Auswahl zwischen verschiedenen Datensprachen nicht möglich ist. Dies führte bei der Sprache *LOTOS* zu einer geringen Akzeptanz, da die Datensprache *ACT ONE* die Bedürfnisse der Anwender zu wenig unterstützt. Außerdem beinhalten die oben erwähnten Ansätze nicht das Konzept der Mobilität, so daß die Erzeugung neuer Kanäle und die Parameterübergabe von Kanälen nicht möglich sind.

In [MPW92] wird gezeigt, daß die Operationen des λ-Kalküls [Bar84] im π-Kalkül durch Parallelität und Kommunikation nachgebildet werden können. Allerdings eignet sich diese Codierung aufgrund ihres geringen Abstraktionsgrads nicht für eine problemorientierte Spezifikation von Daten im π-Kalkül. Der π-Kalkül kennt als einzigen "Datentyp" nur Kanalnamen; eine Verwendung anderer Datentypen ist nicht vorgesehen.

Die Sprachen *Concurrent ML* [Rep89, Rep99] von Reppy und *Facile* von Prasad et al. [GMP89, TLP+93] erweitern die funktionale Sprache *Standard ML* [HMM86, Pau91] um die aus Prozeßalgebren bekannten Konzepte der Prozeßbehandlung und der Kommunikation. In diesen Sprachen sind Daten- und Verhaltensaspekte *symmetrisch* integriert: Prozeßoperationen und Kommunikation sind Bestandteile funktionaler Ausdrücke. Dies führt zu komplexen Semantiken und erschwert die Untersuchung von Äquivalenzbeziehungen von Programmen.

Daten in graphischen Notationen

Im bisherigen Standard für Message Sequence Charts MSC '96 [ITU96a] werden Daten nur informell verwendet; Datenoperationen haben keine formale Semantik und es werden keine Regeln für die Gültigkeitsbereiche von Bezeichnern angegeben. Im neuen Standard MSC 2000 [ITU99] sind Datenoperationen vorgesehen. So ist es möglich, das Binden empfangener Werte an Bezeichner zu spezifizieren sowie Daten bei der Komposition von Diagrammen zu übergeben. Allerdings unterstützen weder die Standardsemantik in [ITU96b] noch die Semantiken für High-level MSC [MR97, GHRW98] die Behandlung von Daten. Eine formale Semantik für MSC 2000 liegt noch nicht vor.

Die *Unified Modeling Language* (UML) unterstützt sowohl die Beschreibung von Daten als auch von Verhalten durch jeweils eigene Diagrammtypen [BRJ98]. Daten können in Form von Klassendiagrammen spezifiziert werden, für die Verhaltensbeschreibung stehen diverse Notationen wie z.B. Statecharts und Interaktionsdiagramme zur Verfügung. Allerdings sind die Beziehungen zwischen diesen Diagrammtypen nicht explizit festgelegt. So bestehen z.B. keine Regeln für Gültigkeitsbereiche von Bezeichnern. Weiterhin wird nicht explizit angegeben, welche Datendeklarationen in welchen Diagrammteilen bekannt sein sollen.

Die Semantik von UML wird mit Hilfe eines Metamodells definiert, das ebenfalls in UML spezifiziert wurde [OMG99]. Diese Semantik legt die Beziehungen zwischen den graphischen Entitäten innerhalb eines Diagramms fest, sie definiert

aber nicht die operationelle Semantik der Diagramme für die Verhaltensspezifikation. Aus diesem Grund ist eine zusätzliche Semantik notwendig, um die von den Diagrammen beschriebenen Abläufe zu ermitteln. Ansätze hierzu werden in [GGW98, GGW99, Kna99, Stö99] vorgestellt. Diese Semantiken konzentrieren sich wiederum auf die Verhaltensaspekte der Diagramme, so daß die Beziehungen zwischen diesen Semantiken und der UML-Semantik für Daten nicht berücksichtigt werden.

Funktionale Programmierung

Wir konzentrieren uns in dieser Arbeit auf die funktionale Spezifikation von Daten. Funktionale Programmierung ist ein Teilgebiet der deklarativen Programmierung [Bac78, Thi94, Rea89] und basiert auf dem λ-Kalkül von Church [Bar84], der eine Formalisierung des Berechenbarkeitsbegriffs auf der Grundlage von Funktionen realisiert. Der Ablauf eines Programms beruht in funktionalen Sprachen nicht auf der sequentiellen Ausführung von Anweisungen, sondern auf der Auswertung von Ausdrücken und der Anwendung von Funktionen auf ihre Argumente. Aufgrund des fehlenden Konzepts von Variablen mit Wertzuweisung gibt es in rein funktionalen Sprachen im Gegensatz zu den imperativen Sprachen keinen Zustandsbegriff.

Aufgrund ihrer Abstraktion von den konkreten Gegebenheiten der Rechnerarchitektur erlauben funktionale Sprachen eine problemorientierte Spezifikation auf hohem Niveau. Weiterhin besitzen funktionale Sprachen in der Regel eine wohldefinierte formale Semantik, die die Verifikation von Programmeigenschaften unterstützt. Diese Semantik ist meist als *Reduktionssemantik* angegeben, die die Auswertung von Ausdrücken definiert. Eine verbreitete Äquivalenz in funktionalen Sprachen ist die β-*Äquivalenz*[Bar84, Thi94]: Zwei Ausdrücke sind äquivalent, wenn sie sich mit der Semantik auf denselben Wert reduzieren lassen.

1.2 Zielsetzung

Ziel der vorliegenden Arbeit ist die Integration von funktionaler Datenbehandlung in Beschreibungstechniken für reaktive Systeme, um eine konsistente Modellierung von Struktur-, Verhaltens- und Datenaspekten zu ermöglichen. Die Konstrukte der Beschreibungstechniken sollen durch eine formale Semantik definiert werden, um die Analysierbarkeit von Systemeigenschaften zu gewährleisten. Weiterhin soll das Konzept der Mobilität zur Spezifikation von Systemen mit dynamischer Struktur berücksichtigt werden.

Um eine problemoriente Beschreibung der Daten zu gewährleisten, soll die Datenintegration nicht nur für eine spezielle funktionale Sprache beschrieben werden. Stattdessen sollen generelle Schnittstellen in die Verhaltenssprachen integriert werden, über die jede funktionale Datensprache, die die Anforderungen

bzgl. der Schnittstellen erfüllt, eingebunden werden kann. Entsprechend soll die Semantik der Verhaltenssprache mit der Reduktionssemantik der jeweiligen Datensprache parametrisiert werden können.

Im Gegensatz zu parallelen funktionalen Sprachen wie *Concurrent ML* [Rep92, Rep99] und *Facile* [GMP89, TLP+93] soll die Integration der beiden Teilsprachen *asymmetrisch* erfolgen: Die Konstrukte der Verhaltenssprache sollen Datenausdrücke zur Berechung ihrer Parameter verwenden können, während die jeweils gewählte Datenteilsprache nicht auf Elemente der Verhaltensteilsprache zugreifen können soll. Auf diese Weise soll die Definition von Semantiken und, damit verbunden, die Angabe von Äquivalenzrelationen auf Spezifikationen wesentlich vereinfacht werden.

Entwicklung einer Prozeßalgebra mit Mobilität und Datenintegration.

Im ersten inhaltlichen Schwerpunkt der Arbeit soll eine Prozeßalgebra definiert werden, die das Konzept der Mobilität mit der Integration von Datenbeschreibung verbindet. Bisherige Prozeßalgebren mit integrierter Datenbeschreibung erlauben nicht die Verwendung von Mobilität, während mobile Prozeßalgebren keine Daten außer den Kanalnamen unterstützen. Daher besteht in dieser Fragestellung ein zentraler Forschungsbedarf. Es ist zum einen zu untersuchen, in welcher Form die Kanalnamen als Teil der Datensprache dargestellt werden können. Weiterhin soll untersucht werden, wie die Standardoperatoren der Prozeßalgebren um die Beschreibung von Datenaspekten erweitert werden können.

Ein weiteres Entwurfsziel ist dabei die Annäherung der Prozeßalgebra an existierende parallele Sprachen wie *Java* [AG96] und *Ada* [Shu88, TDT95] sowie an Bibliotheken zur Prozeßbehandlung für sequentielle Sprachen, um so die Umsetzung von der Spezifikation in ein lauffähiges Programm zu unterstützen. Zu diesem Zweck soll die Algebra statt des üblichen binären Parallelоperators einen unären Operator zur Prozeßerzeugung enthalten.

Erweiterung von Interaktionsdiagrammen um Daten.

Im zweiten Schwerpunkt der Arbeit soll eine Variante von Interaktionsdiagrammen definiert werden, die Elemente für die Beschreibung von Daten beinhaltet. Da in den bisherigen Ansätzen [ITU96a, ITU99, BRJ98] für Verhaltensdiagramme Daten eher informell verwendet werden, soll in unseren Diagrammen eine formale Datenbehandlung integriert werden. Hierzu gehören zum einen die Entwicklung von Regeln für die Definition und Gültigkeit von Bezeichnern, zum anderen soll die Bedeutung der Interaktionsdiagramme durch eine formale Semantik beschrieben werden. Die Interaktionsdiagramme sollen für die Designspezifikation verwendbar sein, daher sollen zum einen komplexe Sprachelemente wie z.B. Schleifen in der Notation enthalten sein, zum anderen soll die Komposition von einzelnen Diagrammen ermöglicht werden, um die Beschreibung des mögli-

chen Systemverhaltens in mehrere, übersichtliche Teilbeschreibungen zerlegen zu können. Dabei soll die Parameterübergabe zwischen Diagrammen möglich sein.

Als Grundlage für die Semantik der Interaktionsdiagramme soll die Prozeßalgebra aus dem ersten Teil der Arbeit verwendet werden, da sie bereits eine formale Notation darstellt, die die Beschreibung von Verhalten und Datentransformationen integriert. Durch die Verwendung der Prozeßalgebra als Semantik für die Diagramme soll somit zugleich die Einsetzbarkeit der Prozeßalgebra als semantische Basis zur Definition anderer Sprachen bzw. Notationen erprobt werden.

1.3 Gliederung

In diesem Abschnitt geben kurz wir eine Übersicht über die einzelnen Kapitel der Arbeit. Die Arbeit ist in zwei Teile gegliedert; der erste Teil behandelt die Entwicklung einer Prozeßalgebra mit Daten, der zweite beinhaltet die Integration von Daten in eine Variante von Interaktionsdiagrammen. Die Semantik der graphischen Sprache des zweiten Teils wird durch eine Übersetzung in die Prozeßalgebra mit Daten des ersten Teils definiert.

In Kapitel 2 führen wir einen unären Operator für Prozeßerzeugung ein, der den üblichen binären Operator für Parallelkomposition ersetzt. Hierfür definieren wir eine einfache Prozeßalgebra $\mathcal{B}$, die weder über Daten noch über Mobilität verfügt. Die Bedeutung der Operatoren geben wir in Form einer operationellen Semantik an. Weiterhin definieren wir die starke Bisimulation als Äquivalenzrelation und weisen durch Anwendung bekannter Ergebnisse nach, daß diese Relation eine Kongruenz für alle Operatoren der Algebra ist.

In Kapitel 3 wird der Kalkül $\mathcal{B}$ um Mobilität und die Integration von Daten erweitert. Dabei wird die Datensprache nicht festgelegt, sondern es werden Kriterien für geeignete Datensprachen definiert. Der erweiterte Kalkül $\mathcal{P}$ betrachtet Kommunikationskanäle als Daten, die dynamisch erzeugt werden und wie Werte anderer Datentypen als Parameter in Nachrichten übertragen und in Datenstrukturen gespeichert werden können. Wir definieren ein Typsystem zur Charakterisierung wohlgetypter Prozeßterme. Die operationelle Semantik von $\mathcal{P}$ ist eine Erweiterung der Semantik von $\mathcal{B}$ um die Behandlung der neuen Sprachkonzepte. Als Äquivalenzen werden die starke und die schwache Bisimulation definiert und es wird gezeigt, daß die starke Bisimulation und eine Variante der schwachen Bisimulation Kongruenzen für alle Operatoren der Sprache sind. Anschließend wird eine axiomatische Semantik für beide Relationen angegeben. Es wird bewiesen, daß die axiomatische Semantik bzgl. der Äquivalenzen korrekt ist. Die Vollständigkeit des axiomatischen Systems wird hingegen nicht untersucht.

Hierauf folgt in Kapitel 4 als Beispiel für die Integration einer funktionalen Datensprache die Definition eines erweiterten, getypten λ-Kalküls. Den so entstandenen Kalkül für Daten- und Verhaltensbeschreibung nennen wir $\mathcal{P}_\lambda$. Als

Typen stehen boolesche Werte, ganze Zahlen, Kommunikationskanäle, Tupel, Listen und Funktionstypen zur Verfügung. Das Typsystem von $\mathcal{P}$ wird um die Typisierung funktionaler Ausdrücke erweitert. Die Semantik der Datensprache wird durch Reduktionsregeln für Ausdrücke definiert. Diese Semantik wird dann zur Reduktion der Datenbestandteile in die Prozeßsemantik integriert. Wir geben zwei Beispiele für die Spezifikation mit $\mathcal{P}_\lambda$: Das erste Beispiel beschreibt eine FIFO-Speicherzelle, das zweite Beispiel ist ein Referenzbeispiel für die Spezifikation paralleler Systeme [BMS96], bei dem der Zugriff auf einen entfernten Speicher über eine *remote procedure call*-Komponente modelliert wird. Das zweite Beispiel enthält auch die Anwendung der axiomatischen Theorie aus Kapitel 3 für den Nachweis der Verhaltensäquivalenz zweier Spezifikationen.

Im zweiten Teil der Arbeit werden als Variante von Interaktionsdiagrammen die *n-Agenten-Diagramme* eingeführt. In Kapitel 5 werden die Konzepte der Diagramme, die Syntax der Diagrammelemente und das zugrundeliegende Systemmodell beschrieben. Wie bei der Prozeßalgebra $\mathcal{P}$ ist die Datensprache nicht vorgegeben, sondern es ist die Integration verschiedener funktionaler Sprachen möglich, die eine kleine Menge von Anforderungen erfüllen. Zur Vereinfachung der Angabe einer Semantik wird eine textuelle Notation definiert, in die die Diagramme überführt werden können. Diese Notation wird durch semantische Regeln ergänzt, die Wohlgeformtheitskriterien für die textuelle Darstellung formulieren. Als Beispiel spezifizieren wir das aus der Spieltheorie bekannte *Gefangenendilemma* [Hof88].

In Kapitel 6 beschreiben wir die Semantik der n-Agenten-Diagramme. Wir definieren eine Menge von Übersetzungsfunktionen, die die textuelle Darstellung von Diagrammen in Prozeßdefinitionen des Kalküls $\mathcal{P}$ aus Kapitel 3 übersetzen. Auf diese Weise wird eine integrierte Semantik sowohl für die Daten- als auch für die Verhaltensaspekte der Diagramme realisiert. Die Übersetzung der Datenbeschreibungen ist von der gewählten Datensprache abhängig. Als Beispiel für eine Integration verwenden wir die Datensprache aus Kapitel 4. Die Übersetzung von Diagrammen wird anhand des Gefangenendilemma-Beispiels sowie eines Client-Server-Beispiels erläutert.

In Kapitel 7 fassen wir die Ergebnisse der Arbeit noch einmal zusammen und führen einen abschließenden Vergleich mit verwandten Ansätzen durch. Weiterhin geben einen Ausblick auf weitere Forschungsarbeiten.

Veröffentlichungen. Eine Variante der Prozeßalgebra $\mathcal{B}$ aus Kapitel 2 wurde in [Geh98] als Grundlage für die Definition der Semantik einer objektbasierten, parallelen Sprache verwendet. Die Inhalte von Kapitel 3 und Kapitel 4 wurden in [GR99] veröffentlicht. Ein Überblick über die n-Agentendiagramme und ihre Semantik erscheint in [Geh00].

Kapitel 2

Prozeßerzeugung

In diesem Kapitel definieren wir eine einfache Prozeßalgebra. Diese Algebra verwendet im Gegensatz zu den herkömmlichen Prozeßalgebren [Hoa85, Mil89, BW90, MPW92] keinen binären Operator für die Parallelkomposition, sondern einen unären Operator für Prozeßerzeugung.

2.1 Syntax

In diesem Abschnitt definieren wir die Syntax unserer Prozeßalgebra. Die elementaren Bestandteile von Termen sind die *Aktionen*, die unteilbare Ausführungsschritte wie z.B. Kommunikation repräsentieren. Aus den Aktionen werden durch Anwendung der Operatoren der Algebra Terme, genannt *Prozesse*, gebildet, die das Verhalten von Systemen beschreiben.

Wie in *CCS* [Mil89] betrachten wir Kommunikation als eine synchrone Aktion zwischen zwei Prozessen, die entsprechende Kommunikationsaktionen über einen gemeinsamen Kanal ausführen. Sei $\mathcal{C}$ eine abzählbare Menge von Kanalnamen, die wir mit $a, b, c, ...$ bezeichnen. Eine Eingabe auf einem Kanal a bezeichnen wir mit $a?$, eine Ausgabe mit $a!$. Wir verwenden † als "Metavariable" zur Darstellung von ? oder !. Die Menge der Kommunikationsaktionen sei $\mathcal{A} = \{a\dagger \mid a \in \mathcal{C}\} \cup \{\tau\}$; Aktionen werden durch α, β bezeichnet. Die *interne Aktion* τ repräsentiert internes Verhalten eines Prozesses und kann nicht mit anderen Aktionen interagieren. Weiterhin sei PROC eine Menge von Prozeßnamen $P, Q, ...$

Wir definieren die Syntax des Basiskalküls $\mathcal{B}$ durch die folgende kontextfreie Grammatik:

$$t := \mathbf{0} \mid \mathbf{1} \mid \alpha \mid \mathit{spawn}(t) \mid t;t \mid t + t \mid \mathbf{ch}\, C.\, t \mid P$$

$\mathbf{0}$ bezeichnet den inaktiven Prozeß, der keine Aktionen ausführen kann. $\mathbf{1}$ bezeichnet einen erfolgreich terminierten Prozeß. Der Operator $\mathit{spawn}(t)$ erzeugt einen neuen Prozeß, der parallel zum erzeugenden Prozeß ausgeführt wird: $\mathit{spawn}(t); u$ repräsentiert die parallele Ausführung von t und u. Mit $t; u$ bezeichnen wir die

sequentielle Komposition von t und u: u kann Aktionen ausführen, wenn t terminiert ist. Der Auswahloperator $t + u$ führt entweder t oder u aus. **ch** $C.\, t$ mit $C \subseteq \mathcal{C}$ ist ein Restriktionsoperator. Der Prozeß t kann nur über Aktionen mit seiner Umgebung interagieren, die nicht die Kanäle in C verwenden. Somit lassen sich die Kanäle in C als lokale Kanäle von t interpretieren. Prozeßnamen $P \in \mathcal{N}$ werden durch eine *Prozeßumgebung* genannte Funktion $\Theta : \textsc{Proc} \to \mathcal{B}$ interpretiert. Für jedes $P \in \textsc{Proc}$ repräsentiert $\Theta(P) = t$ eine Prozeßdeklaration mit dem Namen P und dem Prozeßrumpf t. Wir schreiben $\Theta(P) = t$ auch als $P \mapsto t$.

Aus Gründen der Übersichtlichkeit legen wir Prioritäten für die Operatoren aus $\mathcal{B}$ fest. Wir nehmen an, daß ; die höchste Priorität besitzt, gefolgt von + und danach von **ch** $C.t$. Daher entspricht **ch** $c.\, a! + c!; b!$ dem Term **ch** $c.\, (a! + (c!; b!))$. Weiterhin nehmen wir an, daß die sequentielle Komposition rechtsassoziativ ist, d.h. $t_1; t_2; t_3$ entspricht $t_1; (t_2; t_3)$.

2.2 Semantik

Die Bedeutung der Terme von $\mathcal{B}$ definieren wir durch eine operationelle Semantik. Wir geben eine *Interleaving*-Semantik an, so daß die parallele Ausführung von Prozessen durch eine zeitliche Verzahnung der Aktionen der Prozesse dargestellt wird. Als Modelle für die Semantik verwenden wir beschriftete Transitionssysteme [Plo81].

Definition 2.1 *Ein* beschriftetes Transitionssystem *ist ein Tupel* $(L, S, \to, q)$*, wobei* L *eine Menge von Beschriftungen,* S *eine Menge von Zuständen,* $\to \subseteq S \times L \times S$ *eine Übergangsrelation und* $q \in S$ *der Anfangszustand ist.*

Wir definieren die Menge der Beschriftungen als $\Omega = \mathcal{A} \cup \{\checkmark\}$, wobei $\checkmark$ die Terminierung eines Prozesses signalisiert. Sei $\omega \in \Omega$.

Definition 2.2 *Die* operationelle Semantik *eines Terms* $t \in \mathcal{B}$ *ist das Transitionssystem* $(\Omega, \mathcal{B}, \to, t)$*, wobei die Transitionsrelation* $\to$ *durch die in Abbildung 2.1 angegebenen Transitionsregeln definiert wird.*

Wir definieren hierbei die Funktion[1] $fc : \mathcal{A} \to 2^{\mathcal{C}}$ als $fc(a\dagger) = \{a\}$ und $fc(\tau) = \emptyset$. Die Regel B_1 besagt, daß **1** stets eine Terminierungsaktion ausführen kann. Eine Aktion α führt einen α-Schritt aus und ist anschließend terminiert (Regel B_2). Ein Prozeßname P wird durch den in der Prozeßumgebung Θ definierten Prozeßrumpf $\Theta(P)$ ersetzt. Im Gegensatz zu den üblichen Regeln für Prozeßnamen in Prozeßalgebren führt die Regel B_3 hierbei einen τ-Schritt aus; der Grund hierfür besteht in der Kompatibilität zum Kalkül mit Daten in Kapitel 3. Die Semantik des Auswahloperators $t + u$ wird durch die Regeln B_4 und B_5 festgelegt. Regel

[1] Diese Funktion wird aus Gründen der Kompatibilität mit Kapitel 3 definiert.

$$\frac{}{\mathbf{1} \xrightarrow{\checkmark} \mathbf{1}}\,\mathrm{B}_1 \qquad \frac{}{\alpha \xrightarrow{\alpha} \mathbf{1}}\,\mathrm{B}_2 \qquad \frac{}{P \xrightarrow{\tau} \Theta(P)}\,\mathrm{B}_3$$

$$\frac{t \xrightarrow{\omega} t'}{t + u \xrightarrow{\omega} t'}\,\mathrm{B}_4 \qquad \frac{u \xrightarrow{\omega} u'}{t + u \xrightarrow{\omega} u'}\,\mathrm{B}_5 \qquad \frac{t \xrightarrow{\omega} t' \quad fc(\omega) \cap C = \emptyset}{\mathbf{ch}\ C.\, t \xrightarrow{\omega} \mathbf{ch}\ C.\, t'}\,\mathrm{B}_6$$

$$\frac{}{spawn(t) \xrightarrow{\checkmark} spawn(t)}\,\mathrm{B}_7 \qquad \frac{t \xrightarrow{\alpha} t'}{spawn(t) \xrightarrow{\alpha} spawn(t')}\,\mathrm{B}_8$$

$$\frac{t \xrightarrow{\alpha} t'}{t; u \xrightarrow{\alpha} t'; u}\,\mathrm{B}_9 \qquad \frac{t \xrightarrow{\checkmark} t' \quad u \xrightarrow{\omega} u'}{t; u \xrightarrow{\omega} t'; u'}\,\mathrm{B}_{10}$$

$$\frac{t \xrightarrow{\checkmark} t' \quad t' \xrightarrow{\alpha} t'' \quad u \xrightarrow{\alpha'} u' \quad \{\alpha, \alpha'\} = \{a!, a?\}}{t; u \xrightarrow{\tau} t''; u'}\,\mathrm{B}_{11}$$

Abbildung 2.1: Operationelle Semantik für $\mathcal{B}$.

B_6 legt fest, daß ein Prozeß mit Restriktionsoperator nur Aktionen auf Kanälen durchführen kann, die nicht in der Menge C enthalten sind.

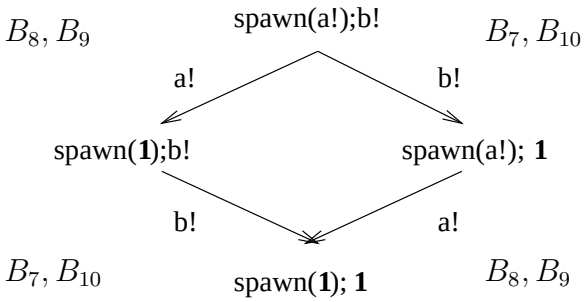
In den Regeln B_7 und B_8 wird die Semantik der Prozeßerzeugung durch den *spawn*-Operator definiert. Regel B_7 legt fest, daß ein $spawn(t)$-Operator jederzeit eine Terminierungstransition durchführen kann, und zwar unabhängig von einer Terminierung von t. In Regel B_8 wird festgelegt, daß $spawn(t)$ alle Transitionen von t mit Ausnahme der Terminierung durchführen kann. Somit kann $spawn(t)$ die Aktionen von t ausführen, obwohl der Prozeß nach Regel B_7 terminiert ist. Diese Eigenschaft ist für die Realisierung der Parallelität von besonderer Bedeutung (siehe unten).

Die Regeln B_9 und B_{10} beschreiben die sequentielle Komposition von Termen. Die Standardregeln für sequentielle Komposition in Prozeßalgebren werden meist wie folgt angegeben (z.B. in [BW90]):

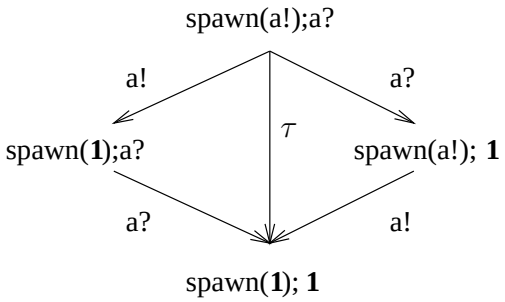
$$\frac{t \xrightarrow{\alpha} t'}{t; u \xrightarrow{\alpha} t'; u} \qquad \frac{t \xrightarrow{\checkmark} t' \quad u \xrightarrow{\omega} u'}{t; u \xrightarrow{\omega} u'}$$

Die erste Regel entspricht unserer Regel B_9. Die zweite Regel ist in unserem Fall nicht verwendbar, da sie den terminierten ersten Operanden t' aus dem Term entfernt. Durch unsere spezielle Behandlung des *spawn*-Operators kann ein terminierter Prozeß jedoch noch Aktionen ausführen, so daß t' nicht aus dem Term entfernt werden darf (Regel B_{10}). Die Regeln für sequentielle Komposition ermöglichen daher in Verbindung mit den Regeln für *spawn* die parallele Ausführung

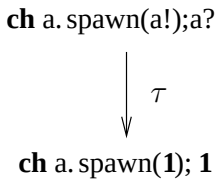
von t und u in $spawn(t); u$. Wir betrachten hierzu das folgende Beispiel:



Regel B_{11} beschreibt die Kommunikation zwischen Prozessen. Wenn der erste Operand t einer sequentiellen Komposition $t; u$ eine Terminierungstransition gefolgt von einer Aktion α ausführen kann, muß t mindestens einen Teilterm der Form $spawn(t_0)$ enthalten. Kann u eine zu α komplementäre Aktion α' ausführen, können t und u sich synchronisieren und eine gemeinsame τ-Aktion durchführen. Die folgende Abbildung enthält das Transitionssystem für einen Prozeß mit Kommunikationsmöglichkeit:



Wird in diesem Beispiel der Kanal a durch einen Restriktionsoperator gebunden, kann gemäß Regel B_6 nur der τ-Schritt der Kommunikation durchgeführt werden:



An diesem Beispiel läßt sich erkennen, daß man durch die Restriktion eines Kanals die Kommunikation auf diesem Kanal erzwingen kann. Somit entspricht die Restriktion eines Kanals einer Einschränkung seines Gültigkeitsbereichs, da eine Kommunikation über diesen Kanal mit anderen Aktionen außerhalb des Restriktionsoperators nicht möglich ist.

Bisimulation. Als Äquivalenz auf Prozessen definieren wir die *starke Bisimulation* [Mil89]. Zwei Prozesse t und u sind *bisimulär*, wenn sie sich gegenseitig in allen Zuständen simulieren können. Kann t eine Transition $t \xrightarrow{\alpha} t'$ ausführen, muß auch u eine entsprechende Transition $u \xrightarrow{\alpha} u'$ ausführen können und die entstandenen Prozesse t' und u' müssen wiederum bisimulär sein.

Definition 2.3 *Sei $R \subseteq \mathcal{B} \times \mathcal{B}$ eine symmetrische Relation. R ist eine* (starke) Bisimulationsrelation*, wenn $(t, u) \in R$ impliziert, daß für alle $t \xrightarrow{\omega} t'$ gilt: $\exists u' : u \xrightarrow{\omega} u'$ und $(t', u') \in R$.*

Zwei Prozesse t und u sind bisimulär, wenn eine Bisimulationsrelation R mit $(t, u) \in R$ existiert. Wir schreiben dann $t \sim u$.

Theorem 2.4 *$\sim$ ist eine Kongruenz für alle Operatoren in $\mathcal{B}$.*

Beweis für Theorem 2.4. Für den Beweis des Theorems verwenden wir ein Resultat aus der Meta-Theorie. Es ist bekannt, daß aus der Kompatibilität der Transitionsregeln einer Semantik mit bestimmten *Formaten* bestimmte Eigenschaften dieser Semantik gefolgert werden können. Hierzu gehört auch die Kongruenzeigenschaft von Äquivalenzrelationen (siehe z.B. [BIM95, JF92]). Aufgrund des Vorkommens von Prädikaten in den Bedingungen der Transitionsregeln verwenden wir das *path*-Format [BV93], das die Verwendung von Prädikaten erlaubt. Es ist leicht zu sehen, daß die Regeln aus Abbildung 2.1 die Bedingungen des *path*-Formats erfüllen; somit ist Bisimulation eine Kongruenz für $\mathcal{B}$. □

2.3 Diskussion

Wir vergleichen den Kalkül $\mathcal{B}$ mit Ansätzen, die ebenfalls statt eines binären Operators für Parallelkomposition einen unären Operator für die Prozeßerzeugung verwenden. Diese Ansätze unterscheiden sich von $\mathcal{B}$ besonders im Hinblick auf die semantische Modellierung der Prozeßerzeugung.

Eine frühe Formalisierung eines Operators für Prozeßerzeugung wurde von Baeten und Vaandrager in [BV92] im Umfeld der Prozeßalgebra *ACP* [BW90] angegeben, die im Gegensatz zu *CCS* [Mil89] und *CSP* [Hoa85] sequentielle Komposition statt Aktionspräfixen verwendet. Während in $\mathcal{B}$ die parallele Ausführung von t und u in $spawn(t); u$ durch die Kombination der Regeln für *spawn* und für sequentielle Komposition erreicht wird, benutzt die in [BV92] vorgestellte Prozeßalgebra *APC* einen Hilfsoperator $t \lceil u$, der eine asymmetrische Parallelkomposition darstellt. Dieser Operator ist asymmetrisch, da die Terminierung des linken Operanden keinen Einfluß auf die Terminierung des Gesamtterms hat; in einem Prozeß $(t_1 \lceil t_2); t_3$ ist die Ausführung von t_3 nur von der Terminierung von t_2 abhängig. Somit entspricht der Prozeß $spawn(t); u$ aus $\mathcal{B}$ direkt dem Prozeß $t \lceil u$ aus *APC*; $\mathcal{B}$ verwendet aber im Gegensatz zu *APC* keinen Hilfsoperator.

Eine weitere Prozeßalgebra mit Prozeßerzeugung ist der *Fork*-Kalkül [Hav94, HL94] von Havelund und Larsen. Zur Modellierung der Prozeßerzeugung wird eine zweistufige Semantik verwendet. Im ersten Schritt wird das Verhalten eines einzelnen Prozesses modelliert. Die so enstandenen Transitionssysteme werden dann zu einem gemeinsamen Transitionssystem kombiniert, das das Verhalten des Gesamtsystems darstellt. Die Regeln für die Prozeßerzeugung lauten:

$$\frac{}{fork(t) \overset{\phi(t)}{\hookrightarrow} \mathbf{1}} \qquad \frac{t \overset{\phi(u)}{\hookrightarrow} t'}{\{|t|\} \xrightarrow{\tau} \{|t', u|\}}$$

Die erste Regel beschreibt die lokale Behandlung eines Prozeßaufrufs. Der Operator für die Prozeßerzeugung signalisiert das Erzeugen eines Prozesses durch eine Transition mit der Beschriftung $\phi(t)$ und ist anschließend terminiert. Die zweite Regel definiert die "globale" Auswirkung einer Prozeßerzeugung. Die globalen Zustände werden durch Multimengen von Prozeßtermen definiert. Führt ein Prozeß eine $\phi(u)$-Aktion aus, wird die Multimenge nach einem τ-Schritt um den neuen Prozeß u erweitert. Die Realisierung der Prozeßerzeugung mittels eines τ-Schritts führt zu Problemen bei der Angabe einer axiomatischen Semantik, da z.B. kein Expansionsgesetz für $fork(a!); b!$ angegeben werden kann; in unserem Kalkül $\mathcal{B}$ gilt hingegen direkt $spawn(a!); b! = a!; b! + b!; spawn(a!)$ (siehe Axiom (3.36) in Abbildung 3.5). Aus diesem Grund wird in [HL94] für die Axiomatisierung ein Hilfsoperator *forked* definiert, der dem *spawn* aus $\mathcal{B}$ entspricht.

Aufgrund seines Operators für Prozeßerzeugung kann der Kalkül $\mathcal{B}$ verwendet werden, um Semantiken für parallele Programmiersprachen und deren Kommunikationsmodelle zu definieren. Eine Semantik für eine abstrakte objektbasierte Sprache wurde in [Geh98] vorgestellt. Das Parallelitätskonzept der dort beschriebenen Sprache basiert auf dem der Skriptsprache *Object REXX* [Mic96, End97]. Die Semantik der Sprache wird durch eine Übersetzung in eine Variante von $\mathcal{B}$ angegeben. Es zeigt sich, daß der unäre Operator für Prozeßerzeugung die Beschreibung des nebenläufigen Verhaltens von Methoden wesentlich vereinfacht, da durch ihn das parallele Verhalten eines Methodenrumpfs direkt in seiner Übersetzung definiert werden kann, ohne daß die Parallelität auf der globalen Systemebene behandelt werden muß.

Kapitel 3

Der Prozeßkalkül

In diesem Kapitel erweitern wir den in Kapitel 2 vorgestellten Kalkül um Mobilität und Daten. Zunächst definieren wir die allgemeinen Anforderungen, die eine Datensprache erfüllen muß, damit sie zur Beschreibung von Daten in Verbindung mit dem Prozeßkalkül verwendet werden kann. Anschließend werden Syntax, Typsystem und operationelle Semantik des Kalküls definiert. Weiterhin definieren wir als Äquivalenzrelationen auf Prozessen die starke und die schwache Bisimulation und charakterisieren beide Relationen durch eine axiomatische Semantik. Da die Beweise für die Theoreme recht umfangreich sind, werden sie nicht im laufenden Text, sondern in einem speziellen Abschnitt 3.7 angegeben.

3.1 Anforderungen an die Datensprache

Der Prozeßkalkül soll nicht auf die Verwendung einer speziellen Datensprache festgelegt sein, sondern es soll die Möglichkeit zur Integration beliebiger Datensprachen bestehen, die eine kleine Anzahl von Annahmen erfüllen. In diesem Abschnitt definieren wir die grundlegenden Begriffe der Datenbeschreibung und formulieren die Anforderungen, die eine Datenteilsprache erfüllen muß.

Sei VAR eine Menge von *Bezeichnern* $x, y, ...$, sei EXPR eine Menge von *Ausdrücken* E, F. Ausdrücke werden durch die Operatoren der jeweils gewählten Datensprache gebildet, so daß wir hier keine allgemeine Definition von Ausdrücken angeben. Ausdrücke können *freie* Bezeichner enthalten, d.h. Bezeichner aus VAR, die nicht durch einen umgebenden Bindungsoperator gebunden sind. Wir bezeichnen die Menge der freien Bezeichner eines Ausdrucks $E \in \text{EXPR}$ mit $fv(E)$ ($\subseteq$ VAR). Ist $fv(E) = \emptyset$, nennen wir E *geschlossen*. Zu den geschlossenen Ausdrücken gehört die Menge VALUE $\subseteq$ EXPR der *Werte*. Werte werden mit $v, v', ...$ bezeichnet und stellen die Konstanten dar, z.B. ganze Zahlen oder Wahrheitswerte, aber auch Funktionsabstraktionen.

Wir nehmen an, daß die Datenteilsprache *getypt* ist. Sei DTYPE eine Menge von Datentypen $T, T', ...$, wobei DTYPE mindestens den Typ *bool* zur Darstel-

lung der Wahrheitswerte *true* und $false$ sowie für jedes $T \in \text{DTYPE}$ einen Typ T *channel*, der Kanäle enthält, die Werte des Typs T übertragen können, enthalten muß. Da T ebenfalls Kanältypen enthalten kann, sind Kanäle typisierbar, über die andere Kanäle als Daten übertragen werden können. Die Verwendung des Typkonstruktors T *channel* erlaubt eine beliebig tiefe Verschachtelung von Kanaltypen, obwohl in Spezifikationen meist eine Tiefe von zwei Verschachtelungen ausreichend ist[1]. Der Grund für die beliebige Verschachtelung besteht in der Gleichbehandlung von Kanälen mit Werten anderer Datentypen; so werden z.B. Typausdrücke für Listen üblicherweise mit einem Typkonstruktor T *list* erzeugt [HMM86].

Die Typisierung von Ausdrücken wird durch *Typzuweisungen* (engl. *type judgements*) der Form $\Delta \vdash E : T$ vorgenommen, wobei Δ eine Menge von *Typannahmen* der Form $x : T$ ist. $x_1 : T_1, ..., x_n : T_n \vdash E : T$ bedeutet, daß unter der Annahme, daß die Bezeichner $x_1, ..., x_n$ die Typen $T_1, ..., T_n$ besitzen, der Ausdruck E den Typ T besitzt. Die x_i dürfen auch frei in E vorkommen. Nicht alle Ausdrücke besitzen Typzuweisungen; Ausdrücke, für die eine Typzuweisung existiert, sind *wohlgetypt*. Die Menge der Werte des Typs T wird mit VALUE^T bezeichnet. Die Menge der Kanalwerte $a, b, c, ...$ nennen wir CHAN. Jeder Kanalwert besitzt einen Typ. Zur Unterstützung der Typisierung erlauben wir eine erweiterte Schreibweise von Kanalwerten, die neben dem Namen des Kanals auch seinen Typ enthält. So ist $c_{\$T\$}$ ein Kanal des Typs T *channel*; wir schreiben $c_{\$T\$} \in \text{CHAN}^T$. Die $c_{\$T\$}$-Notation ist notwendig, da aus einem Kanalnamen nicht syntaktisch sein Typ abgeleitet werden kann, während Werte wie 3 oder *true* allein durch ihre Syntax ihren Typ erkennen lassen. Ist der Typ eines Kanals aus seinem Kontext ersichtlich, kann die explizite Typangabe weggelassen werden. Mit $fc(E)$ benennen wir die Menge der in einem Ausdruck vorkommenden Kanalwerte, die nicht durch einen Restriktionsoperator (s.u.) gebunden sind. Solche Kanalwerte nennen wir *freie Kanalwerte*.

Weiterhin nehmen wir an, daß für die Datenteilsprache eine Reduktionssemantik definiert ist, die einen beliebigen geschlossenen und wohlgetypten Ausdruck E zu einem Wert $[\![E]\!] \in \text{VALUE}$ reduziert. Wenn $\vdash E : T$, dann gilt $[\![E]\!] \in \text{VALUE}^T$, d.h. die Reduktion arbeitet typerhaltend. Basierend auf der Reduktionssemantik nehmen wir an, daß für die Sprache als Äquivalenzrelation eine Art von β-Äquivalenz definiert ist.

Definition 3.1 *Sei für die Datensprache eine Äquivalenzrelation* $=_\beta \subseteq \text{EXPR} \times \text{EXPR}$*, genannt* β*-Äquivalenz, definiert. Es gilt* $E =_\beta F$*, wenn die Ausdrücke* E *und* F *bzgl. der Reduktionssemantik äquivalent sind. Weiterhin nehmen wir an, daß* β*-äquivalente Ausdrücke die gleiche Menge an freien Kanalwerten beinhalten, d.h. aus* $E =_\beta F$ *folgt* $fc(E) = fc(F)$*.*

[1] Zum Beispiel für die Angabe von Kanälen, deren Nachrichten lokale Kanäle für Antworten enthalten (siehe Kapitel 6).

Die Bedingung $fc(E) = fc(F)$ für $E =_\beta F$ ist für die Wohldefiniertheit der operationellen Semantik des Kalküls notwendig.

Wir bezeichnen Vektoren von Bezeichnern, Ausdrücken und Typen mit $\vec{x}$, $\vec{E}$ und $\vec{T}$. Weiterhin bezeichnet $\vec{x} : \vec{T}$ einen Vektor aus Bezeichnern mit Typangaben. Sei $|\vec{x}|$ die Länge eines Vektors $\vec{x}$, sei $\{\vec{x}\}$ die Menge aller Elemente aus $\vec{x}$.

3.2 Syntax

Die Teilsprache $\mathcal{P}$ zur Beschreibung des Systemverhaltens wird durch die folgende kontextfreie Grammatik definiert:

$$t \;::=\; \mathbf{0} \mid \mathbf{1} \mid \tau \mid E!F \mid E?x;t \mid \{\vec{x} \leftarrow \vec{E}\};t \mid t+t \mid t;t \mid \mathbf{ch}\, C.\, t \mid [E] \mid \mathit{spawn}(t) \mid P(\vec{E})$$

Wie in Kapitel 2 bezeichnet **0** den inaktiven Prozeß, der keine Aktionen ausführen kann. **1** bezeichnet einen erfolgreich terminierten Prozeß. Die Aktion τ dient zur Darstellung interner Aktionen eines Prozesses und kann nicht mit anderen Aktionen interagieren. Das Senden von Informationen über einen Kanal wird durch die Aktion $E!F$ dargestellt. E und F sind Ausdrücke aus der Menge EXPR; E berechnet den zu verwendenden Kanal und F den zu übertragenden Wert. $E?x;t$ bezeichnet das Empfangen eines Wertes auf einem Kanal. Nachdem ein Kanal auf dem von dem Ausdruck E definierten Kanal empfangen wurde, wird dieser Wert an den Bezeichner x gebunden. Der Gültigkeitsbereich von x ist der Term t. Der Deklarationsoperator $\{\vec{x} \leftarrow \vec{E}\};t$ deklariert den Vektor $\vec{x}$ von Bezeichnern im Term t; der Ausdrucksvektor $\vec{E}$ definiert die Werte, die an die Bezeichner aus $\vec{x}$ gebunden werden. Der Auswahloperator $t + u$ und der Operator $t;u$ für sequentielle Komposition entsprechen den Operatoren aus Kapitel 2. Der Restriktionsoperator $\mathbf{ch}\, C.t$ definiert eine neue Menge von Kanalwerten, die in t verwendet werden können. Diese Kanäle sind in t lokal, d.h. die Aktionen von t können die Kanäle aus C nicht verwenden, um mit anderen Aktionen außerhalb von t zu interagieren. $[E]$ ist ein bedingter Operator, der nur ausgeführt werden kann, wenn der Ausdruck E zu *true* reduziert wird; andernfalls wird die weitere Ausführung des Prozesses blockiert. Gilt $E = \mathit{true}$, führt $[E]$ einen internen τ-Schritt durch. $\mathit{spawn}(t)$ entspricht dem Operator für Prozeßerzeugung aus Kapitel 2. Deklarationen von Prozessen werden um Parameter erweitert, d.h. Prozeßnamen P aus PROC werden durch eine *Prozeßumgebung* $\Theta : \text{PROC} \to (\text{VAR}^* \times \mathcal{P})$ interpretiert. Für jedes $P \in \text{PROC}$ repräsentiert $\Theta(P) = (\vec{x}, t)$ eine Prozeßdeklaration mit dem Namen P, den formalen Parametern $\vec{x}$ und dem Prozeßrumpf t. Wir schreiben $\Theta(P) = (\vec{x}, t)$ auch als $P(\vec{x}) \mapsto t$. $P(\vec{E})$ beschreibt einen Prozeßaufruf von $\Theta(P)$ mit den aktuellen Parametern $\vec{E}$ (mit $|\vec{E}| = |\vec{x}|$).

Wir verwenden das Symbol ";" auf zwei verschiedene Weisen. Zum einen wird es in den Aktionen $E?x;t$ und $\{\vec{x} \leftarrow \vec{E}\};t$ in Form eines Aktionspräfixes verwendet, zum anderen bezeichnet es die sequentielle Komposition in $t;u$ (mit t beliebig

t	$fv(t)$	$fc(t)$
$\mathbf{0}$	$\emptyset$	$\emptyset$
$\mathbf{1}$	$\emptyset$	$\emptyset$
$E!F$	$fv(E) \cup fv(F)$	$fc(E) \cup fc(F)$
$E?x; u$	$fv(E) \cup (fv(u) \setminus \{x\})$	$fc(E) \cup fc(u)$
$u_1 + u_2$	$fv(u_1) \cup fv(u_2)$	$fc(u_1) \cup fc(u_2)$
$u_1; u_2$	$fv(u_1) \cup fv(u_2)$	$fc(u_1) \cup fc(u_2)$
$[E]$	$fv(E)$	$fc(E)$
$\{\vec{x} \leftarrow \vec{E}\}; u$	$fv(\vec{E}) \cup (fv(u) \setminus \{\vec{x}\})$	$fc(\vec{E}) \cup fc(u)$
$\mathbf{ch}\ C.\ u$	$fv(u)$	$fc(u) \setminus C$
$spawn(u)$	$fv(u)$	$fc(u)$
$P(\vec{E})$	$fv(\vec{E})$	$fc(\vec{E})$

Abbildung 3.1: Freie Bezeichner und freie Kanalwerte.

aus $\mathcal{P}$). Aktionspräfix und sequentielle Komposition sind verwandte Konzepte, so daß eine einheitliche Notation für beide Konzepte die Lesbarkeit der Prozeßterme verbessert.

Analog zu Kapitel 2 definieren wir zur Verbesserung der Lesbarkeit Prioritäten für die Operatoren aus $\mathcal{P}$. Wir nehmen an, daß ; die höchste Priorität besitzt, gefolgt von + und danach von **ch** $C.t$. Daher entspricht **ch** $c.a!c; b!v + b!v'$ dem Term **ch** $c.\,((a!c; b!v) + b!v')$. Weiterhin nehmen wir an, daß die sequentielle Komposition rechtsassoziativ ist, d.h. $t_1; t_2; t_3$ bedeutet $t_1; (t_2; t_3)$. Mit γ bezeichnen wir alle Aktionen, die nach Ausführung terminieren: $\gamma \in \{\tau, E!F, [E] \mid E, F \in \text{Expr}\}$.

In Abbildung 3.1 sind die freien Bezeichner und die freien Kanalwerte eines Terms definiert. Eine Eingabeaktion $E?x; t$ bindet x in t, eine Deklaration $\{\vec{x} \leftarrow \vec{E}\}; t$ bindet $\{\vec{x}\}$ in t. Der Restriktionsoperator **ch** $C.\ t$ bindet die Kanäle der Menge C in t. Für eine Prozeßdefinition $P(\vec{x}) \mapsto t$ fordern wir, daß alle freien Bezeichner in t durch die formalen Parameter von P gebunden werden ($fv(t) \subseteq \{\vec{x}\}$). Weiterhin darf t keine freien Kanäle besitzen ($fc(t) = \emptyset$). Durch diese beiden Forderungen ist gewährleistet, daß zur Ermittlung der freien Bezeichner und der freien Kanäle eines Prozeßaufrufs $P(\vec{E})$ nicht der Prozeßrumpf t berücksichtigt werden muß. Dies erleichtert die Definition von Äquivalenzen auf Termen, da andernfalls die freien Kanäle von t berücksichtigt werden müssen, wobei t wiederum lokale Prozeßaufrufe enthalten kann.

3.3 Typsystem

Wir erweitern das Typsystem in Abschnitt 3.1 angenommene Typsystem der Datensprache in ein Typsystem für $\mathcal{P}$. Der einzige Typ für Prozeßterme ist *proc*, d.h. das Typsystem dient zur Charakterisierung wohlgetypter Prozeßterme.

$$\frac{}{\emptyset \vdash \mathbf{0} : proc}\,\mathrm{T}_1 \qquad \frac{}{\emptyset \vdash \mathbf{1} : proc}\,\mathrm{T}_2 \qquad \frac{}{\emptyset \vdash \tau : proc}\,\mathrm{T}_3 \qquad \frac{\Delta \vdash E : bool}{\Delta \vdash [E] : proc}\,\mathrm{T}_4$$

$$\frac{\Delta \vdash t : proc}{\Delta \vdash spawn(t) : proc}\,\mathrm{T}_5 \qquad \frac{\Delta \vdash E : T\ channel \quad \Delta \vdash F : T}{\Delta \vdash E!F : proc}\,\mathrm{T}_6$$

$$\frac{\Delta \vdash E : T\ channel \quad \Delta, x : T \vdash t : proc}{\Delta \vdash E?x;t : proc}\,\mathrm{T}_7 \qquad \frac{\Delta \vdash \vec{E} : \vec{T} \quad \Delta, \vec{x} : \vec{T} \vdash t : proc}{\Delta \vdash \{\vec{x} \leftarrow \vec{E}\};t : proc}\,\mathrm{T}_8$$

$$\frac{\Delta \vdash t : proc \quad \Delta \vdash u : proc}{\Delta \vdash t + u : proc}\,\mathrm{T}_9 \qquad \frac{\Delta \vdash t : proc \quad \Delta \vdash u : proc}{\Delta \vdash t;u : proc}\,\mathrm{T}_{10}$$

$$\frac{\Delta \vdash \vec{E} : \vec{T} \quad \Theta : P(\vec{x}) \mapsto t \quad \vec{x} : \vec{T} \vdash t : proc \quad fc(t) = \emptyset}{\Delta \vdash P(\vec{E}) : proc}\,\mathrm{T}_{11}$$

$$\frac{\Delta \vdash t : proc}{\Delta \vdash \mathbf{ch}\ C.\,t : proc}\,\mathrm{T}_{12} \qquad \frac{\Delta' \vdash t : proc \quad \Delta' \subseteq \Delta}{\Delta \vdash t : proc}\,\mathrm{T}_{13}$$

Abbildung 3.2: Typsystem für $\mathcal{P}$.

Definition 3.2 *Das Typsystem* TYPE *entsteht aus* DTYPE *durch Hinzufügung des Typs proc und wird durch die Typregeln in Abbildung 3.2 definiert.*

Das Typsystem TYPE für $\mathcal{P}$ ist in Abbildung 3.2 in Form von *Typregeln* angegeben. Die Terme $\mathbf{0}$, $\mathbf{1}$ und τ sind immer wohlgetypt. Ein Bedingungsoperator $[E]$ ist wohlgetypt, wenn der Ausdruck E ein boolescher Ausdruck ist (Regel T_4). Der Term $spawn(t)$ ist wohlgetypt, wenn t wohlgetypt ist (Regel T_5). Regel T_6 legt fest, daß eine Ausgabeaktion wohgetypt ist, wenn der Typ des Kanals und der Typ des zu sendenden Wertes zueinander passen. Eine Eingabeaktion $E?x;t$ ist wohlgetypt, wenn t unter einer zum Typ des Eingabekanals passenden Typannahme für x wohlgetypt ist (Regel T_7). Typregel T_8 behandelt den Fall des Deklarationsoperators analog. Die Terme $t + u$ und $t;u$ sind wohlgetypt, wenn ihre Teilterme wohlgetypt sind (Regeln T_9 und T_{10}). Die Regel T_{11} definiert die bereits in Abschnitt 3.2 beschriebenen Anforderungen für Prozeßdeklarationen. Zum einen muß der Prozeßrumpf durch Typannahmen über die formalen Parameter typisierbar sein, zum anderen darf er keine freien Kanalwerte beinhalten. Zudem müssen bei einem Prozeßaufruf die Typen der aktuellen mit denen der formalen Parameter übereinstimmen. In Regel T_{12} wird definiert, daß die Restriktion von Kanalwerten keine weitere Anforderungen an die Wohlgetyptheit von t stellt, da wir annehmen, daß jeder Kanalwert Element eines entsprechenden Kanaltyps CHAN^T mit $T \in$ DTYPE ist (siehe Abschnitt 3.1). Regel T_{13} dient zur Abschwächung von Typzuweisungen: Ist eine Typzuweisung mit einer Untermenge von Δ möglich, ist sie auch mit Δ möglich.

Bei der Angabe der Syntax des Kalküls in Abschnitt 3.2 haben wir keine Typinformationen für die Bezeichner in $E?x;t$, $\{\vec{x} \leftarrow \vec{E}\};t$ und $P(\vec{x}) \mapsto t$ vorgesehen, da sich die Typen der Bezeichner aus dem Kontext bestimmen lassen (siehe T_7, T_8 und T_{11} in Abbildung 3.2). In den Beispielen in den folgenden Abschnitten werden wir solche zusätzlichen Typinformationen oft angeben, um die Lesbarkeit der Spezifikationen zu verbessern. So werden wir eine Prozeßdefinition oft als $P(\vec{x} : \vec{T}) \mapsto t$ schreiben.

3.4 Operationelle Semantik

Nun definieren wir die operationelle Semantik der Verhaltensteilsprache. Wie für den Basiskalkül in Kapitel 2 geben wir eine *Interleaving*-Semantik an, die die parallele Ausführung von Prozessen durch eine zeitliche Verzahnung der Aktionen der Prozesse darstellt.

Wir verwenden für die Semantik des Kalküls die folgenden Beschriftungen:

- $c?v$ mit $c \in \text{CHAN}$ und $v \in \text{VALUE}$ repräsentiert die Ausführung einer Eingabeaktion, bei der auf dem Kanal c der Wert v gelesen wird.
- $c!v, C$ mit $c \in \text{CHAN}$, $v \in \text{VALUE}$ und $C \subseteq fc(v)$ beschreibt die Ausführung einer Sendeaktion, bei der der Wert v auf dem Kanal c übertragen wird. Dabei wird eine Menge C von lokalen Kanälen, die in v vorkommen, dem Empfänger bekanntgemacht (dies entspricht dem *bound output* im π-Kalkül [MPW92], s.u.). Für $c!v, \emptyset$ schreiben wir auch $c!v$.
- τ repräsentiert die Ausführung einer internen Aktion.
- $\checkmark$ zeigt die erfolgreiche Terminierung eines Prozesses an.

Die Kanalmenge C in der Beschriftung $c!v, C$ mit $C \subseteq fc(v)$ gibt die Menge der Kanäle an, die beim Senden einer Nachricht als Parameter ihrem Gültigkeitsbereich "entkommen". So werden in einem Term $\mathbf{ch}\ c, d, e.\ a!(3, c)$ zuerst die Kanäle c, d und e durch einen Restriktionsoperator als lokale Kanäle deklariert. Anschließend wird c beim Senden einer Nachricht über a als Parameter übertragen. Anschließend ist der Kanal c auch im Empfänger der Nachricht a bekannt, so daß der Gültigkeitsbereich des Kanals durch die Übertragung erweitert wurde (*scope extrusion*). Um diese Erweiterung des Gültigkeitsbereichs in der Semantik abbilden zu können, enthalten Ausgabeaktionen in der Menge C die Menge der jeweils freigegebenen Kanäle. Der obige Term führt eine Transition $\mathbf{ch}\ c, d, e.\ a!(3, c) \xrightarrow{a!(3,c),\{c\}} \mathbf{ch}\ d, e.\ \mathbf{1}$ aus; durch das Übertragen von c wird die lokale Restriktion von c aufgehoben.

Sei Ω die Menge der eben genannten Beschriftungen. Sei $\omega \in \Omega$, sei $\alpha \in \Omega \setminus \{\checkmark\}$.

Zur Definition der Transitionen von $\mathcal{P}$ benötigen wir das Konzept der *Substitution* von Bezeichnern durch Werte. Zu diesem Zweck definieren wir eine partielle Funktion $\sigma : \text{VAR}^* \to \text{VALUE}^*$. Die Anwendung einer Substitution σ auf einen Term t führt zur simultanen Ersetzung der im Definitionsbereich von σ enthaltenen Bezeichner durch die entsprechenden Werte des Wertebereichs. Eine solche Anwendung bezeichnen wir mit $t\sigma$. Soll eine Substitution explizit angegeben werden, verwenden wir die Schreibweise $\langle \vec{v}/\vec{x} \rangle$ mit $\vec{v} \in \text{VALUE}^*$ und $\vec{x} \in \text{VAR}^*$.

Die Semantik eines Terms in $\mathcal{P}$ wird durch Transitionen der Form $t \xrightarrow{\omega} t'$ angegeben. Wir fordern eine zusätzliche Bedingung für solche Transitionen: $t \xrightarrow{c!v,C} t'$ ist nur wohlgeformt, wenn $C \cap fc(t) = \emptyset$ gilt. Dies garantiert, daß lokale Kanäle, die während der Transition durch *scope extrusion* bekanntgemacht werden, nicht schon zuvor frei im Term vorkommen. Die operationelle Semantik eines Terms wird durch die Transitionsregeln in Abbildung 3.3 festgelegt. Zur Umbenennung von Kanalnamen definieren wir die α-Konversion.

Definition 3.3 *Die* α-Konversion *von Kanalnamen definieren wir als Ersetzung eines restringierten Kanalnamens durch einen anderen Kanalnamen gleichen Typs. Wir schreiben die Ersetzung von* $c_{\$T\$}$ *durch* $d_{\$T\$}$ *in einem Term* t *als* $t \ll d_{\$T\$}/c_{\$T\$} \gg$. *Terme* t_1, t_2, *die bis auf die Umbenennung restringierter Kanäle gleich sind, nennen wir* α-äquivalent *(geschrieben als* $t_1 =_\alpha t_2$*). Es gilt* $\mathbf{ch}\ c_{\$T\$}.\ t =_\alpha \mathbf{ch}\ d_{\$T\$}.\ t'$ *mit* $c_{\$T\$}, d_{\$T\$} \in \text{CHAN}^T$, $T \in \text{DTYPE}$ *und* $t' = t \ll d_{\$T\$}/c_{\$T\$} \gg$.

Nun definieren wir die Semantik eines Terms $t \in \mathcal{P}$.

Definition 3.4 *Die* Semantik *eines Terms* $t \in \mathcal{P}$ *ist das Transitionssystem* $(\Omega, \mathcal{P}, \to, t)$, *wobei die Transitionsrelation* $\to$ *durch die in Abbildung 3.3 angegebenen Transitionsregeln sowie die Regel*

$$\frac{t =_\alpha t' \quad t' \xrightarrow{\omega} t''}{t \xrightarrow{\omega} t''}$$

für α*-Konversion definiert wird.*

Die Regel für α-Konversion erlaubt die Umbenennung restringierter Kanalnamen. Somit kann die oben geforderte Eigenschaft $fc(t) \cap C = \emptyset$ für $t \xrightarrow{c!v,C} t'$ eingehalten werden, indem im Bedarfsfall die Kanäle in C zuvor durch α-Konversion in t umbenannt werden.

Im folgenden erläutern wir die Regeln in Abbildung 3.3. In Regel R_1 wird festgelegt, daß **1** jederzeit eine Terminierungsaktion ausführen kann. Eine interne Aktion τ führt einen τ-Schritt aus und wird zu **1** (Regel R_2). Bei der Auswertung eines Bedingungsoperators wird in Regel R_3 zunächst der Ausdruck mit Hilfe der Reduktionssemantik der Datensprache reduziert. Liefert die Auswertung den Wahrheitswert *true*, führt der Operator einen τ-Schritt aus. Andernfalls ist keine

$$\frac{}{\mathbf{1} \xrightarrow{\checkmark} \mathbf{1}}\,\mathrm{R}_1 \qquad \frac{}{\tau \xrightarrow{\tau} \mathbf{1}}\,\mathrm{R}_2 \qquad \frac{[\![E]\!] = true}{[E] \xrightarrow{\tau} \mathbf{1}}\,\mathrm{R}_3 \qquad \frac{}{E!F \xrightarrow{[\![E]\!]![\![F]\!]} \mathbf{1}}\,\mathrm{R}_4$$

$$\frac{}{E?x; t \xrightarrow{[\![E]\!]?v} t\langle v/x\rangle}\,\mathrm{R}_5 \qquad \frac{[\![\vec{E}]\!] = \vec{v} \quad t\langle \vec{v}/\vec{x}\rangle \xrightarrow{\omega} t'}{\{\vec{x} \leftarrow \vec{E}\}; t \xrightarrow{\omega} t'}\,\mathrm{R}_6$$

$$\frac{}{spawn(t) \xrightarrow{\checkmark} spawn(t)}\,\mathrm{R}_7 \qquad \frac{t \xrightarrow{\alpha} t'}{spawn(t) \xrightarrow{\alpha} spawn(t')}\,\mathrm{R}_8$$

$$\frac{t \xrightarrow{\omega} t'}{t + u \xrightarrow{\omega} t'}\,\mathrm{R}_9 \qquad \frac{u \xrightarrow{\omega} u'}{t + u \xrightarrow{\omega} u'}\,\mathrm{R}_{10} \qquad \frac{[\![\vec{E}]\!] = \vec{v} \quad \Theta : P(\vec{x} : \vec{T}) \mapsto t}{P(\vec{E}) \xrightarrow{\tau} t\langle \vec{v}/\vec{x}\rangle}\,\mathrm{R}_{11}$$

$$\frac{t \xrightarrow{\alpha} t'}{t; u \xrightarrow{\alpha} t'; u}\,\mathrm{R}_{12} \qquad \frac{t \xrightarrow{\checkmark} t' \quad u \xrightarrow{\omega} u'}{t; u \xrightarrow{\omega} t'; u'}\,\mathrm{R}_{13}$$

$$\frac{t \xrightarrow{\checkmark} t' \quad t' \xrightarrow{\alpha} t'' \quad u \xrightarrow{\alpha'} u' \quad \{\alpha, \alpha'\} = \{c?v, (c!v, C)\} \quad C \cap fc(t; u) = \emptyset}{t; u \xrightarrow{\tau} \mathbf{ch}\ C.\ t''; u'}\,\mathrm{R}_{14}$$

$$\frac{t \xrightarrow{\omega} t' \quad fc(\omega) \cap C = \emptyset}{\mathbf{ch}\ C.\ t \xrightarrow{\omega} \mathbf{ch}\ C.\ t'}\,\mathrm{R}_{15}$$

$$\frac{t \xrightarrow{c!v,A} t' \quad c \notin C \quad A \cap C = \emptyset \quad fc(v) \cap C \neq \emptyset}{\mathbf{ch}\ C.\ t \xrightarrow{c!v,A \cup (fc(v) \cap C)} \mathbf{ch}\ C \setminus fc(v).\ t'}\,\mathrm{R}_{16}$$

Abbildung 3.3: Operationelle Semantik für $\mathcal{P}$ (wir nehmen für jede Regel zusätzlich an, daß $t \xrightarrow{c!v,C} t'$ nur durchgeführt wird, falls $C \cap fc(t) = \emptyset$ gilt).

Transition möglich. Bei einer Sendeaktion werden in Regel R_4 ebenfalls die Ausdrücke für den Kanal und den zu sendenden Wert reduziert und die Resultate in der Transitionsbeschriftung verwendet. Für eine Eingabeaktion wird zunächst der Ausdruck zur Berechnung des zu verwendenden Kanals reduziert. Dann wird über diesen Kanal ein beliebiger Wert v eingelesen und der Bezeichner x im Prozeß t durch v ersetzt. Analog wird in Regel R_6 die Deklaration von Bezeichnern behandelt. Kann der Ausdrucksvektor $\vec{E}$ in einen Wertevektor $\vec{v}$ reduziert werden und kann t nach der Ersetzung von $\vec{x}$ durch $\vec{v}$ eine Transition durchführen, so kann auch $\{\vec{x} \leftarrow \vec{E}\}; t$ dieselbe Transition durchführen. Eingabeaktionen und Deklarationen sind Präfixoperatoren, daher ist die gesonderte Anzeige der Terminierung der Aktion nicht notwendig.

Die Regeln R_7 und R_8 entsprechen den Regeln B_7 bzw. B_8 aus Kapitel 2.

In den Regeln R_{12}, R_{13} und R_{14} wird die sequentielle Komposition von Termen behandelt. Die Regeln R_{12} und R_{13} entsprechen den Regeln B_9 bzw. B_{10} aus

Kapitel 2. Regel R_{14} beschreibt die Kommunikation zwischen Prozessen. Wenn der erste Operand t in einer sequentiellen Komposition $t; u$ eine Terminierungstransition gefolgt von einer Aktion α ausführen kann, muß er mindestens einen Teilterm der Form $spawn(t_0)$ enthalten. Kann u eine komplementäre Aktion α' durchführen, können sich t und u synchronisieren und eine gemeinsame τ-Aktion durchführen. Da nach der Kommunikation die vom Sender bekanntgemachten lokalen Kanäle in C nun dem Empfänger bekannt sind, wird der Gültigkeitsbereich der Kanäle in C auf Sender und Empfänger erweitert. Dieser Effekt der Kommunikationsregel entspricht der *close*-Regel des π-Kalküls [MPW92]. Ein Term $spawn(\mathbf{ch}\ c.\ a!c; t); a?x; u$ kann somit eine Kommunikationsaktion

$$spawn(\mathbf{ch}\ c.\ a!c; t); a?x; u \xrightarrow{\tau} \mathbf{ch}\ c.\ spawn(\mathbf{1}; t); u\langle c/x\rangle$$

durchführen. Der vorher lokal innerhalb des *spawn*-Operators bekannte Kanal c ist nach der Kommunikation auch außerhalb des *spawn* im Empfänger bekannt, so daß die Restriktion von c nach außen gezogen wurde. Der Restriktionsoperator innerhalb des *spawn* wird durch die Regel R_{16} entfernt (siehe unten).

Die Regeln R_{15} und R_{16} definieren die Semantik des Restriktionsoperators in Verbindung mit Mobilität. Kann ein Prozeß t eine Transition ω ausführen, bei der weder der zu verwendende Kanal in C enhalten ist, noch der übertragende Wert einen Kanal aus C enthält, so kann $\mathbf{ch}\ C.\ t$ denselben Übergang machen (Regel R_{15}). Der Restriktionsoperator bleibt nach dem Übergang unverändert erhalten. Ein Beispiel hierfür ist ein Übergang $\mathbf{ch}\ c.\ a!3 \xrightarrow{a!3} \mathbf{ch}\ c.\ \mathbf{1}$.

Soll über einen restringierten Kanal gesendet werden, ist die Ausführung der Sendeaktion nicht möglich und der Prozeß blockiert: $\mathbf{ch}\ c.\ c!1 \not\longrightarrow$.

Führt t eine Sendeaktion aus, muß untersucht werden, ob der gesendete Wert Kanäle enthält, die in der Menge C restringiert sind. Ist dies nicht der Fall, wird wiederum Regel R_{15} verwendet. Ist die Bedingung hingegen erfüllt, wird ein *bound output* durchgeführt, bei dem die Transitionsbeschriftung die Menge der freigegebenen Kanäle enthält. Im resultierenden Prozeß wird dann die Menge der freigegebenen Kanäle aus dem Restriktionsoperator entfernt (Regel R_{16}). Ein Beispiel hierfür ist $\mathbf{ch}\ c, d.\ a!c \xrightarrow{a!c,\{c\}} \mathbf{ch}\ d.\ \mathbf{1}$. Enthält die Sendeaktion $\xrightarrow{c!v,A}$ bereits eine Menge A bekanntgemachter Kanäle, wird diese Menge um $C \cap fc(v)$ erweitert: $\mathbf{ch}\ c, d.\ \mathbf{ch}\ e.\ a!(c, e) \xrightarrow{a!(c,e),\{c,e\}} \mathbf{ch}\ d.\ \mathbf{1}$. Hierbei ist die Bedingung $A \cap C = \emptyset$ in Regel R_{16} von Bedeutung. Sie gewährleistet die korrekte Behandlung verschachtelter Restriktionen, da ohne diese Regel eine Ableitung $\mathbf{ch}\ c.\ \mathbf{ch}\ c.\ a!c \xrightarrow{a!c,\{c\}} \mathbf{1}$ möglich wäre, was der intuitiven Forderung widerspricht, daß jeder Restriktionsoperator eine neue Menge von Kanälen erzeugt. Zur Einhaltung der Bedingung $A \cap C = \emptyset$ sowie der Wohlgeformtheitsforderung $fc(t) \cap C = \emptyset$ für Ausgabetransitionen $t \xrightarrow{c!v,C} t'$ wenden wir α-Konversion an (siehe die zusätzliche Regel in Definition 3.4). So läßt sich der eben genannte Term $\mathbf{ch}\ c.\ \mathbf{ch}\ c.\ a!c$ durch α-Konversion in einen äquivalenten Term $\mathbf{ch}\ d.\ \mathbf{ch}\ c.\ a!c$ umwandeln, der dann eine Transition $\mathbf{ch}\ d.\ \mathbf{ch}\ c.\ a!c \xrightarrow{a!c,\{c\}} \mathbf{ch}\ d.\ \mathbf{1}$ durchführen kann.

Das folgende Beispiel zeigt die Notwendigkeit der auf Seite 39 genannten Wohlgeformheitsbedingung $C \cap fc(t) = \emptyset$ für Transitionen der Form $t \xrightarrow{c!v,C} t'$:

$$\begin{array}{rl} & spawn((\mathbf{ch}\ c.\ a!c); (\mathbf{ch}\ c.\ b!c)); a?x; b?y; y!x \\ \xrightarrow{\tau} & \mathbf{ch}\ c.\ spawn(\mathbf{1}; (\mathbf{ch}\ c.\ b!c)); b?y; y!c \\ =_\alpha & \mathbf{ch}\ c.\ spawn(\mathbf{1}; (\mathbf{ch}\ d.\ b!d)); b?y; y!c \\ \xrightarrow{\tau} & \mathbf{ch}\ c, d.\ spawn(\mathbf{1}; \mathbf{1}); d!c \end{array}$$

In diesem Beispiel wird der Kanalname c für zwei verschiedene lokale Kanalwerte verwendet. Die Wohlgeformheitsbedingung gewährleistet nun, daß der zweite Kanal durch α-Konversion umbenannt wird, bevor er der Umgebung bekanntgemacht wird. Andernfalls würde ein Konflikt mit der Bekanntmachung des ersten Kanals c entstehen.

Das folgende Theorem zeigt, daß, wenn ein wohlgetypter Prozeß eine Transition durchführt, der resultierende Prozeß ebenfalls wohlgetypt ist. Somit ist gewährleistet, daß alle Prozesse, die bei der Ausführung eines wohlgetypten Prozesses auftreten, ebenfalls wohlgetypt sind. Dieses Theorem entspricht dem *subject reduction*-Theorem von Curry (z.B. in [Bar90]).

Theorem 3.5 *Wir nehmen an, daß* $\vdash t : proc$ *gilt.*

- *Wenn* $t \xrightarrow{c?v} t'$, *dann* $\vdash c : T$ *channel für ein* $T \in$ DType*; wenn auch* $\vdash v : T$, *dann* $\vdash t' : proc$.
- *Wenn* $t \xrightarrow{c!v,C} t'$, *dann* $\vdash c : T$ *channel und* $\vdash v : T$ *für ein* $T \in$ DType *und* $\vdash t' : proc$.
- *Wenn* $t \xrightarrow{\omega} t'$ *und* $\omega = \tau$ *oder* $\omega = \checkmark$, *dann* $\vdash t' : proc$.

3.5 Bisimulation und Axiomatisierung

In diesem Abschnitt definieren wir als Äquivalenzrelationen über $\mathcal{P}$ die *starke* und die *schwache* Bisimulation (siehe [Mil89]). Beide Relationen basieren auf der Idee, daß zwei Systeme als äquivalent betrachtet werden können, wenn sie dieselben Transitionen ausführen können. Im Unterschied zur starken Bisimulation wird bei der schwachen Relation von der Ausführung interner Aktionen abstrahiert.

3.5.1 Starke Bisimulation

Gemäß der Standard-Definition der starken Bisimulation in [Mil89] und Definition 2.3 in Kapitel 2 sind zwei Prozesse t und u bisimulär, wenn sie in der Lage sind, Transitionen $t \xrightarrow{\omega} t'$ und $u \xrightarrow{\omega} u'$ auszuführen und die resultierenden Prozesse

t' und u' ebenfalls bisimulär sind. Die Forderung, daß beide Prozesse denselben ω-Übergang ausführen können müssen, ist für unser Konzept der Integration einer Datensprache zu restriktiv. Wenn wir zum Beispiel eine Datensprache in $\mathcal{P}$ integrieren, die auf dem λ-Kalkül [Bar90] beruht, würden die Transitionsbeschriftungen $a!(\lambda x.\lambda y.x + y)$ und $a!(\lambda x.\lambda y.y + x)$ unterschieden werden, obwohl die übertragenen λ-Abstraktionen für alle Argumente das jeweils selbe Resultat liefern. Daher fordern wir nicht die Identität der Transitionsbeschriftungen, sondern verlangen, daß die Beschriftungen *β-äquivalent* im Sinne der in Abschnitt 3.1 erläuterten Relation $=_\beta$ sind. Zuerst erweitern wir den Begriff der β-Äquivalenz auf Transitionsbeschriftungen.

Definition 3.6 *Die Relation $=_\beta$ wird auf Transitionsbeschriftungen wie folgt definiert:*

- $\tau =_\beta \tau$
- $\checkmark =_\beta \checkmark$
- $c?v =_\beta c?w$ *für* $v =_\beta w$
- $(c!v, C) =_\beta (c!w, C)$ *für* $v =_\beta w$

Weiterhin ist $\langle \vec{v}/\vec{x} \rangle =_\beta \langle \vec{v}'/\vec{x} \rangle$ *für* $\vec{v} =_\beta \vec{v}'$.

Nun können wir die starke Bisimulation auf Prozessen definieren.

Definition 3.7 *Sei $R \subseteq \mathcal{P} \times \mathcal{P}$ eine symmetrische Relation. R ist eine* Bisimulationsrelation*, wenn $(t, u) \in R$ impliziert, daß für alle $t \xrightarrow{\omega} t'$ gilt:*

- *Wenn $\omega = c?v$, dann $\forall \omega', \omega =_\beta \omega', \exists u' : u \xrightarrow{\omega'} u'$ und $(t', u') \in R$.*
- *Wenn $\omega \neq c?v$, dann $\exists \omega', \omega =_\beta \omega', \exists u' : u \xrightarrow{\omega'} u'$ und $(t', u') \in R$.*

Zwei geschlossene Terme t, u sind *bisimulär*, wenn eine Bisimulationsrelation R mit $(t, u) \in R$ existiert. Wir schreiben dann $t \sim_\beta u$. Sind t und u offene Terme, fordern wir Bisimulation unter allen Substitutionen: $t \sim_\beta u$ falls $\forall \sigma : t\sigma \sim_\beta u\sigma$.

Die Relation entspricht der *early bisimulation* des π-Kalküls [MPW92]. Für Eingabetransitionen $t \xrightarrow{c?v} t'$ fordern wir, daß u eine entsprechende Transition für jeden Wert v', der zu v äquivalent ist, durchführen kann, da die nachfolgenden Terme unter allen Eingaben bisimulär sein müssen. Für Ausgabetransitionen fordern wir nur, daß eine äquivalente Transition $u \xrightarrow{\omega'} u'$ existiert.

Theorem 3.8 *$\sim_\beta$ ist eine Kongruenz für alle Operatoren aus $\mathcal{P}$.*

Das folgende Lemma besagt, daß die Anwendung äquivalenter Substitutionen auf denselben Term t zu bisimulären Termen führt.

Lemma 3.9 *Wenn $\sigma =_\beta \sigma'$, dann gilt $t\sigma \sim_\beta t\sigma'$.*

3.5.2 Axiomatisierung

Wir definieren eine axiomatische Semantik für $\mathcal{P}$, die die starke Bisimulation charakterisiert. In Abbildung 3.4 und Abbildung 3.5 ist die axiomatische Theorie $\mathcal{AX}_{\sim_\beta}$ für die starke Bisimulation angegeben. Die Axiome lassen sich wie folgt in Gruppen einteilen:

- Die Axiome (3.1) bis (3.12) beschreiben den Teil des Kalküls ohne Kanalerzeugung und Prozeßgenerierung. Mit Ausnahme des Bedingungs- und des Deklarationsoperators ist dieser Teil des Kalküls identisch mit *BPA*, dem Teil der Prozeßalgebra *ACP* ohne Parallelkomposition [BW90]. Daher entsprechen diese Axiome Standardaxiomen anderer Algebren.

- Die Axiome (3.13) bis (3.24) beschreiben die Interaktion des Restriktionsoperators mit den anderen Operatoren aus $\mathcal{P}$.

- Axiom (3.25) beschreibt, daß ein Prozeßaufruf einem internen Schritt gefolgt von der Ausführung des Prozeßrumpfes entspricht, bei der die formalen Parameter durch die aktuellen Parameters des Aufrufs ersetzt worden sind. Wir fordern hierbei, daß die aktuellen Parameter zu Werten reduziert wurden; andernfalls wäre es möglich, daß die Reduzierung der Parameter zu Werten nicht terminiert und somit die linke Seite des Axioms den Prozeßaufruf nicht durchführen könnte, während die rechte Seite einen τ-Schritt ausführen könnte.

- Die Axiome (3.26) bis (3.29) behandeln die Prozeßerzeugung mit *spawn*.

- Mit den Axiomen (3.30) bis (3.35) wird das Zusammenspiel der Datenteilsprache, insbesondere in bezug auf die β-Äquivalenz, mit der Prozeßteilsprache beschrieben. Sie besagen im wesentlichen, daß zwei β-äquivalente Datenausdrücke in einem Prozeßterm ausgetauscht werden können.

- Axiom (3.36) ist das *Expansionsgesetz* für unseren Kalkül. Es beschreibt, daß Terme mit Parallelität in äquivalente Terme mit Auswahloperatoren umgeformt werden können. Zum Beispiel gilt

  $$spawn(a!b); a?x; \mathbf{1} = a!b; a?x; \mathbf{1} + a?x; spawn(a!b) + \tau$$

  (unter Anwendung weiterer Axiome aus Abbildung 3.4).

Zur übersichtlichen Darstellung des Expansionsgesetzes verwenden wir die Notation der *abgeleiteten Ausgabeaktionen*, die den *abgeleiteten Präfixen* aus [MPW92] entsprechen. Falls $c \notin D$ gilt, können wir einen Term $\mathbf{ch}\ D.\, c!v; t$ durch $(c!v, C); t'$ mit $C = D \cap fc(v)$ und $t' = \mathbf{ch}\ D \setminus fc(v).\, t$ ersetzen. Der Teilterm $(c!v, C)$ heißt

$$
\begin{array}{rcll}
[false] & = & \mathbf{0} & (3.1)\\
[true] & = & \tau & (3.2)\\
\mathbf{1};t & = & t & (3.3)\\
t;\mathbf{1} & = & t & (3.4)\\
\mathbf{0};t & = & \mathbf{0} & (3.5)\\
t_1;(t_2;t_3) & = & (t_1;t_2);t_3 & (3.6)\\
\{\vec{x}\leftarrow\vec{v}\};t & = & t\langle\vec{v}/\vec{x}\rangle & (3.7)\\
t+\mathbf{0} & = & t & (3.8)\\
t+u & = & u+t & (3.9)\\
t_1+(t_2+t_3) & = & (t_1+t_2)+t_3 & (3.10)\\
t+t & = & t & (3.11)\\
(t_1+t_2);t_3 & = & t_1;t_3+t_2;t_3 & (3.12)
\end{array}
$$

$$
\begin{array}{rcll}
\mathbf{ch}\ C.\,\mathbf{0} & = & \mathbf{0} & (3.13)\\
\mathbf{ch}\ C.\,\mathbf{1} & = & \mathbf{1} & (3.14)\\
\mathbf{ch}\ C.\,spawn(t) & = & spawn(\mathbf{ch}\ C.\,t) & (3.15)\\
\mathbf{ch}\ C.\,t & = & t \quad \text{für } C\cap fc(t)=\emptyset & (3.16)\\
\mathbf{ch}\ C.\,\mathbf{ch}\ C'.\,t & = & \mathbf{ch}\ C\cup C'.\,t & (3.17)\\
\mathbf{ch}\ C.\,t+u & = & \mathbf{ch}\ C.\,t+\mathbf{ch}\ C.\,u & (3.18)\\
\mathbf{ch}\ C.\,c?x;t & = & \mathbf{0} \quad \text{für } c\in C & (3.19)\\
\mathbf{ch}\ C.\,c!v;t & = & \mathbf{0} \quad \text{für } c\in C & (3.20)\\
\mathbf{ch}\ C.\,\gamma;t & = & \gamma;\mathbf{ch}\ C.\,t \quad \text{für } C\cap fc(\gamma)=\emptyset & (3.21)\\
\mathbf{ch}\ C.\,c?x;t & = & c?x;\mathbf{ch}\ C.\,t \quad \text{für } c\notin C & (3.22)\\
(\mathbf{ch}\ C.\,t);u & = & \mathbf{ch}\ C.\,t;u \quad \text{für } C\cap fc(u)=\emptyset & (3.23)\\
t;\mathbf{ch}\ C.\,u & = & \mathbf{ch}\ C.\,t;u \quad \text{für } C\cap fc(t)=\emptyset & (3.24)\\
P(\vec{v}) & = & \tau;t\langle\vec{v}/\vec{x}\rangle \quad \text{für } \Theta: P(\vec{x})\mapsto t & (3.25)
\end{array}
$$

$$
\begin{array}{rcll}
spawn(\mathbf{0}) & = & \mathbf{1} & (3.26)\\
spawn(t);spawn(u) & = & spawn(u);spawn(t) & (3.27)\\
spawn(t);spawn(u) & = & spawn(spawn(t);u) & (3.28)\\
spawn(t_1;spawn(t_2)+t_3) & = & spawn(t_1;t_2+t_3) & (3.29)
\end{array}
$$

$$
\begin{array}{rcll}
E!F & = & E'!F' \quad \text{für } E=_\beta E',\ F=_\beta F' & (3.30)\\
E?x;t & = & E'?x;t \quad \text{für } E=_\beta E' & (3.31)\\
\{\vec{x}\leftarrow\vec{E}\};t & = & \{\vec{x}\leftarrow\vec{E}'\};t \quad \text{für } \vec{E}=_\beta \vec{E}' & (3.32)\\
[E] & = & [E'] \quad \text{für } E=_\beta E' & (3.33)\\
P(\vec{E}) & = & P(\vec{E}') \quad \text{für } \vec{E}=_\beta \vec{E}' & (3.34)\\
t\langle v/x\rangle & = & t \quad \text{für } x\notin fv(t) & (3.35)
\end{array}
$$

Abbildung 3.4: Axiome für starke Bisimulation.

Sei $\delta \in \{(c!v, C), [E], \tau \mid E \in \text{EXPR}, c \in \text{CHAN}, C \subseteq \text{CHAN}\}$.

$$
\begin{array}{lrcl}
\text{Wenn} & t & = & \sum_{i\in\mathcal{I}} \delta_i; t_i + \sum_{j\in\mathcal{J}} E_j?x_j; t_j \\
\text{und} & u & = & \sum_{k\in\mathcal{K}} \delta_k; t_k + \sum_{l\in\mathcal{L}} E_l?x_l; t_l \\
\text{und} & & & \forall j \in \mathcal{J} : x_j \notin fv(u) \text{ and } \forall l \in \mathcal{L} : x_l \notin fv(t), \\
\text{dann} & \mathit{spawn}(t); u & = & \sum_{i\in\mathcal{I}} \delta_i; \mathit{spawn}(t_i); u + \sum_{k\in\mathcal{K}} \delta_k; \mathit{spawn}(t); u_k \\
& & + & \sum_{j\in\mathcal{J}} E_j?x_j; \mathit{spawn}(t_j); u + \sum_{l\in\mathcal{L}} E_l?x_l; \mathit{spawn}(t); u_l \\
& & + & \sum_{\substack{i\in\mathcal{I}, l\in\mathcal{L} \\ \delta_i=(c!v,C), [\![E_l]\!]=c}} \tau; \mathbf{ch}\ C.\ \mathit{spawn}(t_i); u_l\langle v/x_l\rangle \\
& & + & \sum_{\substack{j\in\mathcal{J}, k\in\mathcal{K} \\ [\![E_j]\!]=c, \delta_k=(c!v,C)}} \tau; \mathbf{ch}\ C.\ \mathit{spawn}(t_j\langle v/x_j\rangle); u_k
\end{array}
\tag{3.36}
$$

Abbildung 3.5: Expansionsgesetz

dann *abgeleitete Ausgabeaktion*. Durch die Anwendung der abgeleiteten Ausgabeaktionen als Hilfsnotation können wir schrittweise die Restriktionsoperatoren aus einem Term entfernen.

Das Expansionsgesetz besitzt eine Nebenbedingung, die für die korrekte Behandlung von Termen mit freien Bezeichnern notwendig ist. Ohne diese Bedingung wären Gleichungen wie

$$\mathit{spawn}(a?x; \mathbf{1}); b!x = (a?x; \mathit{spawn}(\mathbf{1}); b!x) + (b!x; \mathit{spawn}(a?x; \mathbf{1}))$$

ableitbar, bei der x auf der linken Seite in $b!x$ der Gleichung frei vorkommt, während es auf der rechten Seite in einem Teilterm durch $a?x$ gebunden wird. Daher müßte vor einer Anwendung des Expansionsgesetzes der Bezeichner x in $a?x$ durch α-Konversion in einen anderen Bezeichner umgewandelt werden.

Lassen sich zwei Terme $t, u \in TS$ durch die Gleichungen der Theorie $\mathcal{AX}_{\sim_\beta}$ als äquivalent beweisen, schreiben wir $\mathcal{AX}_{\sim_\beta} \vdash t = u$. Das folgende Theorem zeigt die Korrektheit der axiomatischen Theorie $\mathcal{AX}_{\sim_\beta}$ bzgl. $\sim_\beta$.

Theorem 3.10 *Die axiomatische Theorie $\mathcal{AX}_{\sim_\beta}$ ist* korrekt *bezüglich $\sim_\beta$: Für $\mathcal{AX}_{\sim_\beta} \vdash t = u$ gilt $t \sim_\beta u$.*

Wir untersuchen nicht die Vollständigkeit von $\mathcal{AX}_{\sim_\beta}$ bzgl. $\sim_\beta$. Der Nachweis der Vollständigkeit geschieht i. allg. durch die Angabe einer Normalform, in die alle Terme des Kalküls übertragen werden können [Mil89, PS95]. In dieser Normalform ist Parallelität durch Anwendung des Expansionsgesetzes (vergleiche Abbildung 3.5) durch eine Auswahl der möglichen *interleavings* der beteiligten Aktionen ersetzt worden, so daß Terme in Normalform keine Parallelkomposition

mehr enthalten. Außerdem werden die Operatoren für die Kanalrestriktion aus dem Term entfernt (hierzu werden im π-Kalkül die abgeleiteten Ausgabeaktionen verwendet [PS95]). Liegen die Terme in Normalform vor, wird ein Induktionsbeweis über die Tiefe, d.h. die maximalen Folgen von Transitionen zweier semantisch äquivalenter Terme in Normalform geführt, in dem gezeigt wird, daß es für jede Alternative einer Auswahl eines Terms eine entsprechende Alternative einer Auswahl im jeweils anderen Term gibt. Auf diese Weise läßt sich induktiv über die Tiefe zeigen, daß äquivalente Terme durch das axiomatische Systeme schrittweise als gleich identifiziert werden.

Während das Finden einer geeigneten Normalform für einfache Prozeßalgebren wie *CCS* relativ einfach ist [Mil89], wäre dies für unseren Kalkül aufgrund von sequentieller Komposition, Mobilität und Kanalerzeugung sehr aufwendig. Hinzu kommt, daß durch den unären *spawn*-Operator die Anwendung des Expansionsgesetzes das *spawn* nicht aus dem Term entfernt wird ($spawn(a!1); b!2 = a!1; b!2 + b!2; spawn(a!1)$), so daß sich die *spawn*-Operatoren durch mehrfache Anwendung des Gesetzes nur bis in den letzten Term einer sequentiellen Komposition "schieben" lassen. Aufgrund der zu erwartenden komplexen Normalform für die Terme aus $\mathcal{P}$ und den damit verbundenen hohen Aufwand für den Nachweis der Vollständigkeit haben wir auf den Vollständigkeitsbeweis verzichtet, insbesondere, da die Vollständigkeit für den praktischen Einsatz des axiomatischen Systems zur Analyse von Termen von eher untergeordneter Bedeutung ist.

Das folgende Korollar enthält Beispiele für Gleichungen, die aus den Axiomen von $\mathcal{AX}_{\sim_\beta}$ abegeleitet wurden.

Korollar 3.11 *Folgende Gleichungen sind Beispiele für aus $\mathcal{AX}_{\sim_\beta}$ abgeleitete Gleichungen:*

$$\begin{aligned} spawn(spawn(t)) &= spawn(t), \\ spawn(\mathbf{1}) &= \mathbf{1}, \\ spawn(t_1; spawn(t_2)) &= spawn(t_1; t_2). \end{aligned}$$

3.5.3 Schwache Bisimulation

Die starke Bisimulation abstrahiert nicht von den internen Schritten von Prozessen, daher ist sie als Äquivalenzrelation auf Prozeßtermen in einigen Fällen zu stark unterscheidend. Aus diesem Grund wurde als weitere Äquivalenzrelation die *schwache Bisimulation* eingeführt, die nur die sichtbaren Aktionen von Prozessen betrachtet [Mil89].

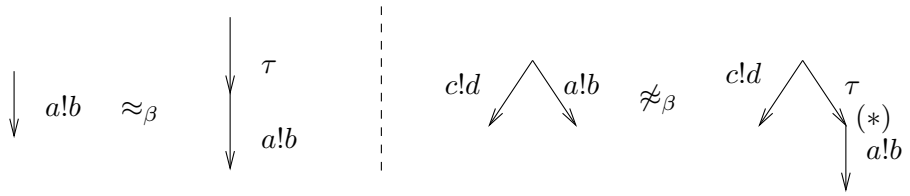
Sei $\Rightarrow$ die reflexive und transitive Hülle von $\xrightarrow{\tau}$. Mit $\stackrel{\omega}{\Longrightarrow}$ bezeichnen wir $\Rightarrow\xrightarrow{\omega}\Rightarrow$, also eine Folge von Transitionen, bei der vor und nach dem ω-Schritt beliebig viele τ-Transitionen auftreten können (eventuell auch keine). Weiterhin definieren wir $\stackrel{\widehat{\omega}}{\Longrightarrow}$ als $\Rightarrow$ für $\omega = \tau$ und als $\stackrel{\omega}{\Longrightarrow}$ andernfalls. Somit kann $\stackrel{\widehat{\tau}}{\Longrightarrow}$ auch die leere Folge von τ-Transitionen sein, während $\stackrel{\tau}{\Longrightarrow}$ mindestens einen τ-Schritt ausführt.

Definition 3.12 *Sei $R \subseteq \mathcal{P}\times\mathcal{P}$ eine symmetrische Relation. R ist eine* schwache Bisimulationsrelation*, wenn $(t,u) \in R$ impliziert, daß für alle Transitionen $t \xrightarrow{\omega} t'$ die folgenden Bedingungen gelten:*

- *Wenn $\omega = c?v$, dann $\forall\omega', \omega =_\beta \omega', \exists u' : u \stackrel{\widehat{\omega'}}{\Longrightarrow} u'$ und $(t',u') \in R$.*
- *Wenn $\omega \neq c?v$, dann $\exists\omega', \omega =_\beta \omega', \exists u' : u \stackrel{\widehat{\omega'}}{\Longrightarrow} u'$ und $(t',u') \in R$.*

Zwei geschlossene Terme t, u sind *schwach bisimulär*, wenn eine schwache Bisimulationsrelation R mit $(t,u) \in R$ existiert. Wir schreiben $t \approx_\beta u$. Sind t und u offene Terme, fordern wir schwache Bisimulation unter allen Substitutionen: $t \approx_\beta u$ falls $\forall\sigma : t\sigma \approx_\beta u\sigma$.

Das folgende Theorem besagt, daß schwache Bisimulation für alle Operatoren von $\mathcal{P}$, mit Ausnahme des Auswahloperators, eine Kongruenz ist. So gilt $a!b \approx_\beta \tau; a!b$, aber $c!d + a!b \not\approx_\beta c!d + \tau; a!b$ (vergleiche [Mil89]):



Die beiden Transitionssysteme auf der rechten Seite sind nicht bisimulär, da im Zustand $(*)$ keine $\xrightarrow{c!d}$-Transition möglich ist.

Theorem 3.13 *$\approx_\beta$ ist eine Kongruenz bzgl. spawn, Definition von Bezeichnern, Eingabeaktionen, Kanalrestriktion und sequentieller Komposition.*

Neben den in Abbildung 3.4 und Abbildung 3.5 angegebenen Axiomen für starke Bisimulation, die ebenso für schwache Bisimulation korrekt sind, lassen sich weitere Axiome für die Behandlung von τ-Schritten angeben, die nur für schwache Bisimulation korrekt sind.

Proposition 3.14 *Für alle $t \in \mathcal{P}$ gilt: $\tau; t \approx_\beta t$ und $spawn(\tau; t) \approx_\beta spawn(t)$. Weiterhin gilt $\mathbf{1}; \tau \approx_\beta \mathbf{1}$.*

Die mangelnde Kongruenzeigenschaft bzgl. des Auswahloperators ist aus *CCS* bekannt [Mil89]. Die dort beschriebene Lösung läßt sich auch auf unseren Kalkül anwenden. Wir definieren hierzu den Begriff der *schwachen Kongruenz.*

Definition 3.15 *Sei $R \subseteq \mathcal{P} \times \mathcal{P}$ eine symmetrische Relation. R ist eine* schwache Kongruenzrelation *(engl.* rooted bisimulation *oder* weak congruence*), wenn $(t,u) \in R$ impliziert, daß für alle Transitionen $t \xrightarrow{\omega} t'$ die folgenden Bedingungen gelten:*

$\gamma;\tau$	$=$	γ	(3.37)	$t+\tau;t$	$=$	$\tau;t$	(3.39)
$spawn(\tau;t)$	$=$	$\tau;spawn(t)$	(3.38)	$E?x;\tau;t$	$=$	$E?x;t$	(3.40)

Abbildung 3.6: Zusätzliche Axiome für schwache Kongruenz.

- *Wenn* $\omega = c?v$, *dann* $\forall\omega', \omega =_\beta \omega', \exists u' : u \stackrel{\omega'}{\Longrightarrow} u'$ *und* $t' \approx_\beta u'$.
- *Wenn* $\omega \neq c?v$, *dann* $\exists\omega', \omega =_\beta \omega', \exists u' : u \stackrel{\omega'}{\Longrightarrow} u'$ *und* $t' \approx_\beta u'$.

Zwei geschlossene Terme t, u sind *schwach kongruent*, geschrieben $t \simeq_\beta u$, wenn eine schwache Kongruenzrelation R mit $(t, u) \in R$ existiert.

Die schwache Kongruenzrelation ist eine stärkere Variante der schwachen Bisimulation, bei der gefordert wird, daß jede *initiale* τ-Transition vom jeweils anderen Term durch eine entsprechende τ-Transition simuliert werden können muß (Verwendung von $\stackrel{\omega}{\Longrightarrow}$ statt $\stackrel{\widehat{\omega}}{\Longrightarrow}$ im ersten Ableitungsschritt, in weiteren Schritten ist nur die schwache Bisimulation gefordert). Für die auf den ersten Schritt folgenden Transitionen besteht kein Unterschied zwischen schwacher Kongruenz und schwacher Bisimulation.

Die folgende Proposition beschreibt die Beziehung zwischen starker Bisimulation, schwacher Bisimulation und schwacher Kongruenz.

Proposition 3.16 $t \sim_\beta u$ *impliziert* $t \simeq_\beta u$, *und* $t \simeq_\beta u$ *impliziert* $t \approx_\beta u$.

Theorem 3.17 $\simeq_\beta$ *ist die größte in* $\approx_\beta$ *enthaltene Kongruenz für die Operatoren aus* $\mathcal{P}$.

In Abbildung 3.6 erweitern wir die axiomatische Theorie $\mathcal{AX}_{\sim_\beta}$ um Axiome für schwache Kongruenz. Wir bezeichnen die erweiterte Theorie mit $\mathcal{AX}_{\simeq_\beta}$. Die Gleichungen in Proposition 3.14 sind hingegen nicht korrekt für schwache Kongruenz; sie zeigen den Unterschied zwischen schwacher Bisimulation und schwacher Kongruenz.

Analog zum Korrektheitsnachweis für $\mathcal{AX}_{\sim_\beta}$ in Theorem 3.10 zeigen wir im folgenden Theorem die Korrektheit von $\mathcal{AX}_{\simeq_\beta}$ bzgl. $\simeq_\beta$.

Theorem 3.18 *Die axiomatische Theorie* $\mathcal{AX}_{\simeq_\beta}$ *ist korrekt bezüglich* $\simeq_\beta$*: Für* $\mathcal{AX}_{\simeq_\beta} \vdash t = u$ *gilt* $t \simeq_\beta u$.

Auch hier untersuchen wir nicht die Vollständigkeit, da der Beweis sehr komplex wäre und Vollständigkeit für die Einsetzbarkeit von $\mathcal{AX}_{\simeq_\beta}$ von eher untergeordneter Bedeutung ist.

3.6 Diskussion

In diesem Kapitel haben wir den Kalkül $\mathcal{P}$ eingeführt, der einen ersten Ansatz für eine Prozeßalgebra darstellt, die sowohl Mobilität als auch Datenbeschreibung unterstützt. Hierzu wurde der Kalkül $\mathcal{B}$ aus Kapitel 2 um deklarative Datenbehandlung und Mobilität erweitert. Dabei wurde nicht die Verwendung einer bestimmte Datensprache festgelegt, sondern es kann jede funktionale Sprache verwendet werden, die die in Abschnitt 3.1 beschriebenen Anforderungen erfüllt. P verwendet den unären Operator *spawn* zur Prozeßerzeugung. Die operationelle Semantik von Prozeßtermen wurde durch die Angabe von Transitionsregeln definiert. Als Äquivalenzrelationen auf Prozessen wurden sowohl die starke als auch die schwache Bisimulation als auch die schwache Kongruenzrelation (*rooted bisimulation*) für $\mathcal{P}$ definiert. Für diese Relationen wurden axiomatische Semantiken angegeben und deren Korrektheit bewiesen.

Im folgenden diskutieren wir verwandte Ansätze für mobile Prozeßkalküle bzw. Kalküle mit Daten.

Bindungen und sequentielle Komposition. Die Operatoren von $E?x;t$, $\{\vec{x} \leftarrow \vec{E}\};t$ und **ch** $C.\,t$ von $\mathcal{P}$, die Bindungen erzeugen, sind als Präfixoperatoren realisiert. Dies hat den Vorteil, daß für jeden definierten Bezeichner x der Gültigkeitsbereich t durch die Syntax von $\mathcal{P}$ vorgegeben wird. Hierdurch wird festgelegt, in welchem Teilterm ein definierter Bezeichner durch den an ihn gebundenen Wert substituiert werden soll. Wären die bindungserzeugenden Sprachkonstrukte stattdessen wie die anderen Operatoren mit den nachfolgenden Operationen durch sequentielle Komposition verbunden, wäre der Bereich für die Ersetzung nicht fest vorgegeben, so daß die Substitution sukzessive auf alle folgenden Terme angewendet werden müßte. Weiterhin müßte ein Mechanismus entwickelt werden, der die Substitution an die nachfolgenden Terme weiterreicht. Hierdurch wird zum einen die Semantik aufwendiger, zum anderen müssen Äquivalenzrelationen wie die Bisimulation die noch "aktiven" Substitutionen berücksichtigen. Untersuchungen hierzu wurden in [GR97] durchgeführt. Der dort definierte mobile Kalkül $\mathcal{F}$ verwendet Substitutionen als spezielles syntaktisches Konstrukt. Die Terminierung von Prozessen wird als Prädikat angegeben, das zusätzlich Informationen über noch durchzuführende Substitutionen beinhaltet. Es wird eine starke Bisimulation für $\mathcal{F}$ angegeben, die aber bzgl. der Durchführung von Substitution keine Kongruenzrelation ist. Daher haben wir in $\mathcal{P}$ die bindungserzeugenden Operatoren als Präfixoperator realisiert.

π-Kalkül. Der Standardkalkül für die Beschreibung von Systemen mit Mobilität ist der π-Kalkül [MPW92] von Milner, Parrow und Walker. In diesem Kalkül wird Kommunikation durch Kanalnamen realisiert. Diese Kanalnamen werden auf zwei Weisen verwendet. Zum einen repräsentieren sie die Kommunikationskanäle,

über die andere Namen übertragen werden können; somit werden sie als *Werte* interpretiert. Zum anderen werden sie auch als *Bezeichner* verwendet, die bei der Eingabe auf Kanälen durch Substitution ersetzt werden können.

Durch diese zweifache Verwendung von Kanalnamen als Bezeichner bzw. Werte gibt es im π-Kalkül keine offenen Terme. Daher ist Bisimulation bzgl. Substitution und damit auch für Eingabeaktionen keine Kongruenz. Es gilt[2]

$$\begin{array}{rrcl} & \overline{x}z.\mathbf{0} \mid y(w).\mathbf{0} & \sim & \overline{x}z.y(w).\mathbf{0} + y(w).\overline{x}z.\mathbf{0} \ , \\ \text{aber} & a(y).(\overline{x}z.\mathbf{0} \mid y(w).\mathbf{0}) & \not\sim & a(y).(\overline{x}z.y(w).\mathbf{0} + y(w).\overline{x}z.\mathbf{0}), \end{array}$$

da der anfänglich auf dem Kanal a gelesene Name auch x sein darf und somit in diesem Fall der linke Term nach der Substitution von y durch x eine Kommunikationsaktion durchführen kann, die für den rechten Term nicht möglich ist [MPW92]. In unserem Kalkül $\mathcal{P}$ gibt es durch die strikte Trennung von Bezeichnern und Werten offene Terme. Offene Terme können erst dann ausgeführt werden, wenn alle ihre freien Bezeichner durch Werte substituiert worden sind. Da offene Terme nur bisimulär sind, wenn sie unter allen Substitutionen bisimulär sind, tritt das eben beschriebene Problem des π-Kalküls in unserem Kalkül nicht auf.

Da Kanalnamen die einzigen "Daten" des π-Kalküls sind, ist eine problemorientierte Darstellung von Datentransformationen nicht möglich. Es gibt aber Ansätze, Daten und die auf ihnen definierten Operationen in die Operationen des π-Kalküls zu übersetzen, siehe z.B. [Mil90, ALT95, Wal95, RS99]. Da die Datenoperationen durch Kanalerzeugung, parallele Prozesse und Kommunikation nachgebildet werden müssen, ergeben sich bei der Übersetzung meist komplexe π-Kalkül-Terme, was die Anwendung von Analyse- und Verifikationsverfahren erschwert.

Im *polyadischen π-Kalkül* [Mil93] ist es möglich, in einer Kommunikationsaktion mehrere Werte über einen Kanal zu übertragen bzw. zu empfangen. So werden in der Aktion

$$\overline{c}de.t \mid c(xy).u \xrightarrow{\tau} t \mid u\langle d/x, e/y\rangle$$

die Namen d und e über c übertragen. In $\mathcal{P}$ läßt sich polyadische Kommunikation durch die Übertragung von Datentupeln realisieren. So läßt sich der obige Term[3] des polyadischen π-Kalküls als

$$spawn(c!(d,e); t'); c?val; \{x \leftarrow \#1\ val, y \leftarrow \#2\ val\}; u'$$

darstellen. Im Empfänger wird das gelesene Tupel (d, e) an den Bezeichner val gebunden. Anschließend werden die beiden Komponenten des Tupels an die Bezeichner x bzw. y gebunden[4]. Dies setzt voraus, daß die gewählte Datensprache die Behandlung von Tupeln unterstützt.

[2] Im π-Kalkül bezeichnet $x(y).t$ eine Eingabeaktion auf x und $\overline{x}y.t$ eine Ausgabe auf x.
[3] Wie in $\mathcal{P}$ verwenden wir t und u für Prozeßterme anstelle von P und Q aus [Mil93].
[4] Die Funktion $\#i$ greift auf die i-te Komponente eines Tupels zu (siehe Kapitel 4).

Für den π-Kalkül existieren mehrere Varianten der Bisimulationsäquivalenz. Sie unterscheiden sich bzgl. des Zeitpunkts, an dem die Substitution von Namen in Termen vorgenommen wird. Bei der *early bisimulation* werden die Terme *nach* der Substitution betrachtet. Dies hat zur Folge, daß alle Terme, die aus den ursprünglichen Termen durch Substitution hervorgehen können, auf die Eigenschaft der Bisimilarität untersucht werden müssen. Bei der *late bisimulation* werden die Terme *vor* der Substituion betrachtet, so daß die Substitutionen in die Bisimulationsrelation integriert werden. Daher werden die ursprünglichen Terme nicht verändert und die Ersetzung von Namen findet nur in der Relation statt. Dies ist möglich, da der π-Kalkül keine offenen Terme kennt und daher auch auch Terme ohne Ersetzung von Namen ausführbar sind. Axiomatische Semantiken für die beiden Arten der Bisimulation sind in [PS95] angegeben.

Der $\mathcal{P}$-Kalkül hat eine Semantik, bei der Bezeichner durch Werte ersetzt werden. Daher entspricht die Bisimulation in Definition 3.7 der *early bisimulation* des π-Kalküls. Um eine *late*-Relation zu definieren, müsste zuvor die operationelle Semantik offener Terme definiert werden. Dies könnte durch die Angabe einer Semantik geschehen, bei ein Zustand aus einem Term und einer *Umgebung* besteht, die den freien Bezeichnern der Terme Werte zuordnet.

ψ-Kalkül. Der ψ-Kalkül [Hav94] von Havelund und Larsen ist eine Erweiterung des zuvor in Abschnitt 2.3 behandelten *Fork*-Kalküls [Hav94, HL94]. Wie der π-Kalkül verwendet der ψ-Kalkül nur Kanalnamen, so daß auch hier eine problemorientierte Darstellung von Datenoperationen nicht möglich ist. Da der ψ-Kalkül auf dem *Fork*-Kalkül basiert, wird die Semantik von Termen ebenfalls durch ein zweistufiges Transitionssystem definiert. Hierbei entsteht als zusätzliche Schwierigkeit die Verwaltung der erzeugten Kanäle, so daß die Zustände des globalen Transitionssystems auch eine Menge der bisher bekannten Kanalnamen enthalten. Dies erschwert die Angabe einer axiomatischen Semantik, da bei der Definition der Gleichungen die sich dynamisch zur Laufzeit ändernde Kanalmenge berücksichtigt werden muß.

Fusion calculus. Im π-Kalkül existieren zwei Operatoren zur Erzeugung von Bindungen: die Eingabeaktion $a(x).t$ und die Restriktion eines Namens $(x)t$. Hierdurch entsteht eine Unsymmetrie zwischen der Eingabe- und der Ausgabeaktion, da letztere keine Bindung erzeugt.

Der *Fusion Calculus* von Parrow und Victor [PV98] trennt die Bindung von Bezeichnern von den Eingabeaktionen. In Kommunikationsaktionen werden keine Namen übertragen, sondern es werden Äquivalenzklassen über Namen, sogenannte *fusions*, gebildet. Der obige Beispielterm aus dem polyadischen π-Kalkül lautet im Fusion Calculus wie folgt:

$$\overline{c}de.t \mid cxy.u \xrightarrow{\{d=x,e=y\}} t \mid u$$

Nach der Transition findet keine Substitution im Empfänger statt, sondern die *fusion* $\{d = x, e = y\}$ in der Beschriftung der Transition zeigt die Äquivalenz der Namen d und x bzw. e und y an. Die Zuordnung der Namen zu Äquivalenzklassen geschieht unabhängig von Eingabe- oder Ausgabe, so daß sich für den folgenden Term, bei dem die Eingabe- und Ausgabe vertauscht wurde, die folgende Transition ergibt:

$$cde.t \mid \overline{c}xy.u \xrightarrow{\{d=x,e=y\}} t \mid u$$

Obwohl Eingabe und Ausgabe vertauscht wurden, ist die *fusion* in der Transition zum vorhergehenden Beispiel identisch.

Der Fusion Calculus verwendet einen globalen Zustandsbegriff, d.h. die Äquivalenz von Namen ist nicht nur im Empfänger, sondern stets im ganzen Prozeßterm bekannt. Daher wird im Term

$$\overline{c}de.t \mid cxy.u \mid t' \mid u' \xrightarrow{\{d=x,e=y\}} t \mid u \mid t' \mid u'$$

die Äquivalenz der Namen auch in den "unbeteiligten" Komponenten t' und u' bekannt. Die Einschränkung des Gültigkeitsbereichs einer *fusion* geschieht durch den *scoping*-Operator $(x)t$, der den Namen x in t zu einem lokalen Namen macht. Mit der Einschränkung des Gültigkeitsbereichs für x und y ergibt sich

$$\overline{c}de.t \mid ((x)(y)cxy.u) \mid t' \mid u' \xrightarrow{\mathbf{1}} t \mid u\{d/x, e/y\} \mid t' \mid u',$$

wobei **1** im Fusion Calculus die "leere" *fusion* bezeichnet, die alle Namen auf sich selbst abbildet.

Als Äquivalenzrelation verwendet der Fusion Calculus die sogenannte *Hyper-Äquivalenz*, die eine Bisimulation darstellt, die unter allen möglichen Substitutionen in Termen gelten muß. Der Grund hierfür ist, daß im Fusion Calculus jederzeit durch die Angabe von Namensäquivalenzen Subsitutionen auf Kanalnamen vorgenommen werden können. Durch die Hyper-Äquivalenz entfällt die Unterscheidung der verschiedenen Bisimulationsvarianten des π-Kalküls.

Wie im π-Kalkül besitzt der Fusion Calculus nur Kanalnamen als Daten, so daß eine problemorientierte Datenbehandlung nicht möglich ist. Um eine Übertragung unseres Konzepts aus $\mathcal{P}$ für die Integration von Daten in den Fusion Calculus zu realisieren, ist eine Modifikation der *fusions* notwendig: Die *fusions* definieren dann nicht die Äquivalenz von Namen definieren, sondern repräsentierenim Sinne einer *Umgebung* die Bindung von Werten an Bezeichner:

$$c!(1,2); t \mid c?x; u \xrightarrow{\{x=(1,2)\}} t \mid u$$

Entsprechend muß auch die Definition von Bezeichnern mittels $\{\vec{x} \leftarrow \vec{E}\}; t$ eine Transition mit der Beschriftung $\{\vec{x} = \vec{E}\}$ durchführen, um die neuen Bindungen bekanntzugeben. Weiterhin muß der *scoping*-Operator zur Beschränkung des Gültigkeitsbereichs von Bezeichnern verwendet werden, da der Restriktionsoperator **ch** $C.t$ aus $\mathcal{P}$ sich nur auf Werte bezieht. Da aufgrund der Unterscheidung von Bezeichnern und Werten in $\mathcal{P}$ offene Terme entstehen können, ist die Integration der Umgebungen in die Zustände der operationellen Semantik notwendig.

LOTOS. Die Sprache *LOTOS* (Language of Temporal Ordering Specification) [BB87, ISO87] wurde von der ISO als Standardsprache für die formale Spezifikation verteilter Systeme entwickelt. Sie verwendet das Kommunikationsmodell der Prozeßalgebra *TCSP* [BHR84], in dem nicht zwischen Sende- und Empfangsaktion unterschieden, sondern die Kommunikation durch das gemeinsame Ausführen von Aktionen realisiert wird. Hierzu wird im Parralleloperator eine Menge von Aktionen, genannt *Synchronisationsmenge*, angegeben, über die die Prozesse synchronisieren müssen. So läßt sich folgende Transition ableiten:

$$a; b; stop \parallel_{\{a\}} a; c; stop \xrightarrow{a} b; stop \parallel_{\{a\}} c; stop$$

Nach Ausführung der Transition können nun die Aktionen b und c unabhängig voneinander ausgeführt werden, da über sie nicht synchronisert werden muß. Diese Art der Synchronisation erlaubt, im Gegensatz zu den bisher diskutierten Kalkülen, die Synchronisation von mehr als zwei Prozessen.

LOTOS erlaubt die Beschreibung von Daten mit der algebraischen Spezifikationssprache *ACT ONE* [Man88a, Man88b]. Datenparameter in den Aktionen werden als Teil des Aktionsnamens interpretiert. Daher entstehen bei Verwendung unendlicher Datenbereiche, wie z.B. den natürlichen Zahlen, unendliche Mengen von Aktionsnamen.

Eingaben werden in der Form $a?x : T; B(x)$ angegeben, wobei T der Wertebereich des zu lesenden Werts ist. $B(x)$ ist der Folgeterm, der den Gültigkeitsbereich von x definiert. Da das Kommunikationsmodell von CSP nicht zwischen Eingabe und Ausgabeaktion unterscheidet, wird die Eingabe durch eine Auswahl zwischen allen möglichen Ausgabeaktionen simuliert. Eine Eingabe $a?x : T; B(x)$ wird durch eine Auswahl **choice** $x : T[]a!x; B(x)$ simuliert, die für jeden möglichen Wert des Typs T eine Alternative enthält. Daher entspricht die Eingabe $a?x : nat; B(x)$ der Auswahl $a!1; B(1) + a!2; B(2) + ...$ In der Synchronisationsmenge werden implizit alle Aktionen a durch die um Werte erweiterten Namen $a!1, a!2, ...$ erweitert, so daß folgende Transition ausgeführt wird:

$$a?x : nat; B(x) \parallel_{\{a\}} a!42; B \xrightarrow{a!42} B(42) \parallel_{\{a\}} B.$$

In LOTOS sind Kanalnamen keine Daten, so daß Kanäle nicht in Kommunikationsaktionen übertragen werden können. Somit unterstützt LOTOS nicht die aus den zuvor disktutierten Kalkülen bekannte Mobilität. Die Integration eines Mobilitätskonzepts in LOTOS ist wegen der verwendeten CSP-Kommunikation aufwendig, da sich die Substitutionen empfangener Kanäle sich auch auf die Synchronisationsmengen der Parralleloperatoren auswirken müssen. In [Wat97] wird ein Ansatz zur Darstellung der Mobilitätsapekte des π-Kalküls in LOTOS vorgestellt. Dieser Ansatz verwendet zusätzliche Datenparameter in den Aktionen, um das Kommunikationsmodell von CCS bzw. dem π-Kalkül nachzubilden. Weiterhin werden die Gültigkeitsbereiche sowie das Erweitern des Gültigkeitsbereichs eines lokalen Kanals durch Kommunikation (*scope extrusion*) durch spezielle Prozesse

verwaltet, die die Kommunikation über restringierte Kanäle steuern. Auf diese Weise lassen sich zwar mobile Kanäle in LOTOS simulieren; allerdings erschwert der hierfür notwendige Aufwand die problemorientierte Spezifikation mobiler Systeme.

Die neue Version *Extended LOTOS* (E-LOTOS) [ISO97] verwendet eine funktionale Beschreibung der Daten von Systemen. Die Datensprache ist Bestandteil der Prozeßsprache; Funktionen werden als spezielle Prozesse realisiert, die deterministisches Verhalten aufweisen müssen. Durch diese Realisierung ist die Verwendung von Funktionen höherer Ordnung nicht möglich, so daß viele Standardverfahren aus der Funktionalen Programmierung (*Currying*, Funktionen als Werte, Unterversorgung mit Argumenten usw. [Rea89, Thi94]) nicht anwendbar sind (siehe auch Abschnitt 4.4). Durch die Trennung von Daten- und Prozeßteilsprache in $\mathcal{P}$ ist hier die Verwendung höherer Funktionen möglich (siehe Kapitel 4), so daß die Standardverfahren der funktionalen Programmierung direkt eingesetzt werden können. Auch in E-LOTOS ist Mobilität nicht vorgesehen, so daß sich Systeme mit dynamischer Struktur nicht spezifizieren lassen.

3.7 Beweise

Theorem 3.5. Wir nehmen an, daß $\vdash t : \mathit{proc}$ gilt.

- Wenn $t \xrightarrow{c?v} t'$, dann $\vdash c : T\ \mathit{channel}$ für ein $T \in \text{DTYPE}$; wenn auch $\vdash v : T$, dann $\vdash t' : \mathit{proc}$.
- Wenn $t \xrightarrow{c!v,C} t'$, dann $\vdash c : T\ \mathit{channel}$ und $\vdash v : T$ für ein $T \in \text{DTYPE}$ und $\vdash t' : \mathit{proc}$.
- Wenn $t \xrightarrow{\omega} t'$ und $\omega = \tau$ oder $\omega = \checkmark$, dann $\vdash t' : \mathit{proc}$.

Für den Beweis von Theorem 3.5 benötigen wir das folgende Lemma:

Lemma 3.19 *Wenn $\Delta, \vec{x} : \vec{T} \vdash t : \mathit{proc}$ und $\emptyset \vdash \vec{v} : \vec{T}$, dann gilt $\Delta \vdash t\langle \vec{v}/\vec{x} \rangle : \mathit{proc}$.*

Beweis für Lemma 3.19. Durch Induktion über die Termstruktur von t. Substitutionen distribuieren über alle Operatoren von $\mathcal{P}$. Alle freien Vorkommen von $\vec{x}$ in t werden durch Werte desselben Typs ersetzt. Daher können die Typannahmen $\vec{x} : \vec{T}$ aus Δ entfernt werden. Für den Fall $t = \mathbf{ch}\ C.\ u$ mit $fc(\vec{v}) \cap C \neq \emptyset$ ändern wir die Kanalwerte in C und u durch α-Konversion. □

Beweis für Theorem 3.5. Wir nehmen an, daß sowohl $\vdash t : \mathit{proc}$ als auch $t \xrightarrow{\omega} t'$ gilt und beweisen $\vdash t' : \mathit{proc}$ für die Regeln der operationellen Semantik in Abbildung 3.3.

$\mathbf{R}_1$ Dann gilt $t = \mathbf{1}$, $\omega = \checkmark$ und $t' = \mathbf{1}$. Mit T_2, T_{13} wissen wir, daß $\vdash \mathbf{1} : \mathit{proc}$ gilt. Daher gilt auch $\vdash t' : \mathit{proc}$.

$\mathbf{R}_2$ Dann gilt $t = \tau$, $\omega = \tau$ und $t' = \mathbf{1}$. Mit T_2, T_{13} wissen wir, daß $\vdash t' : \mathit{proc}$ gilt.

$\mathbf{R}_3$ Dann gilt $t = [E]$, $[\![E]\!] = \mathit{true}$, $\omega = \tau$ und $t' = \mathbf{1}$. Weiterhin wissen wir mit T_2, T_{13}, daß $\vdash t' : \mathit{proc}$ gilt.

$\mathbf{R}_4$ Dann gilt $t = E!F$, $\omega = [\![E]\!]![\![F]\!]$ und $t' = \mathbf{1}$. Mit T_6 wissen wir, daß $\vdash E : T\ \mathit{channel}$ und $\vdash F : T$ gilt. Weiterhin können wir mit T_2, T_{13} folgern, daß $\vdash t' : \mathit{proc}$ gilt.

$\mathbf{R}_5$ Dann gilt $t = E?x;t$, $\omega = [\![E]\!]?v$ und $t' = t\langle v/x \rangle$. Mit T_7 wissen wir, daß $\vdash E : T\ \mathit{channel}$ und $x : T \vdash t : \mathit{proc}$ gilt. Dann können wir mit Lemma 3.19 und $\vdash v : T$ schließen, daß $\vdash t' : \mathit{proc}$ gilt.

$\mathbf{R}_7$ Dann gilt $t = \mathit{spawn}(u)$, $\omega = \checkmark$ und $t' = \mathit{spawn}(u)$. Mit T_5 wissen wir, daß $\vdash t' : \mathit{proc}$ gilt.

R_8 Dann gilt $t = spawn(u)$, $u \xrightarrow{\alpha} u'$ und $t' = spawn(u')$. Wir treffen eine adäquate Unterscheidung für α[5]. Mit T_5 und $\vdash spawn(u) : proc$ wissen wir, daß $\vdash u : proc$ gilt. Durch Anwendung der Induktionshypothese erhalten wir $\vdash u' : proc$. Daher können wir mit T_5 folgern, daß $\vdash spawn(u') : proc$ gilt.

R_6 Dann gilt $t = \{\vec{x} \leftarrow \vec{E}\}; u$, $[\![\vec{E}]\!] = \vec{v}$, $u\langle\vec{v}/\vec{x}\rangle \xrightarrow{\omega} u'$ und $t' = u'$. Wir treffen eine adäquate Unterscheidung für ω. Mit T_8 können wir folgern, daß $\vdash \vec{E} : \vec{T}$ und $\vec{x} : \vec{T} \vdash u : proc$ gilt. Hieraus können wir mit $[\![\vec{E}]\!] = \vec{v}$ und Lemma 3.19 schließen, daß $\vdash u\langle\vec{v}/\vec{x}\rangle : proc$ gilt. Daher erhalten wir nach Anwendung der Induktionshypothese $\vdash u' : proc$.

R_9 Dann gilt $t = t_1 + t_2$ und $t_1 \xrightarrow{\omega} t'$. Wir treffen eine adäquate Unterscheidung für ω. Mit T_9 und $\vdash t_1 + t_2 : proc$ können wir ableiten, daß $\vdash t_1 : proc$ gilt. Daher erhalten wir nach Anwendung der Induktionshypothese $\vdash t' : proc$.

R_{10} Analog zum Beweis für R_9.

R_{11} Dann gilt $t = P(\vec{E})$, $[\![\vec{E}]\!] = \vec{v}$, $P(\vec{x} : \vec{T}) \mapsto u \in \Theta$, $\omega = \tau$ und $t' = u\langle\vec{v}/\vec{x}\rangle$. Mit T_{11} wissen wir, daß $\vdash \vec{E} : \vec{T}$, $fc(t) = \emptyset$ und $\vec{x} : \vec{T} \vdash u : proc$ gilt. Daher können wir mit $[\![\vec{E}]\!] = \vec{v}$ und Lemma 3.19 folgern, daß $\vdash u\langle\vec{v}/\vec{x}\rangle : proc$ gilt.

R_{12} Dann gilt $t = t_1; t_2$, $t_1 \xrightarrow{\alpha} t_1'$, $\omega = \alpha$ und $t' = t_1'; t_2$. Wir treffen eine adäquate Unterscheidung für α. Mit T_{10} wissen wir, daß $\vdash t_1 : proc$ und $\vdash t_2 : proc$ gilt. Daher erhalten wir nach Anwendung der Induktionshypothese $\vdash t_1' : proc$. Dann wissen wir mit T_{10}, daß $\vdash t_1'; t_2 : proc$ gilt.

R_{13} Dann gilt $t = t_1; t_2$, $t_1 \xrightarrow{\checkmark} t_1'$, $t_2 \xrightarrow{\omega} t_2'$ und $t' = t_1'; t_2'$. Wir treffen eine adäquate Unterscheidung für ω. Mit T_{10} wissen wir, daß $\vdash t_1 : proc$ und $\vdash t_2 : proc$ gilt. Dann können wir durch Anwendung der Induktionshypothese $\vdash t_1' : proc$ und $\vdash t_2' : proc$ folgern. Dann können wir mit T_{10} schließen, daß $\vdash t_1'; t_2' : proc$.

R_{14} Dann gilt $t = t_1; t_2$, $t_1 \xrightarrow{\checkmark} t_1'$, $t_1' \xrightarrow{\alpha} t_1''$, $t_2 \xrightarrow{\alpha'} t_2'$, $\{\alpha, \alpha'\} = \{c?v, (c!v, C)\}$, $\omega = \tau$ und $t' = \mathbf{ch}\ C.\ t_1''; t_2'$. Mit T_{10} wissen wir, daß $\vdash t_1 : proc$ und $\vdash t_2 : proc$ gilt. Daher erhalten wir nach Anwendung der Induktionshypothese $\vdash t_1' : proc$, $\vdash t_1'' : proc$ und $\vdash t_2' : proc$. Dann können wir mit T_{10} folgern, daß $\vdash t_1''; t_2' : proc$ gilt. Hieraus folgt mit $T_{12} \vdash \mathbf{ch}\ C.\ t_1''; t_2' : proc$.

R_{15} Dann gilt $t = \mathbf{ch}\ C.\ u$, $u \xrightarrow{\omega} u'$, $fc(\omega) \cap C = \emptyset$ und $t' = \mathbf{ch}\ C.\ u'$. Wir treffen eine adäquate Unterscheidung für ω. Mit T_{12} wissen wir, daß $\vdash u : proc$ gilt. Nach Anwendung der Induktionshypothese erhalten wir $\vdash u' : proc$. Daher können wir mit T_{12} folgern, daß $\vdash \mathbf{ch}\ C.\ u' : proc$ gilt.

[5]Wir untersuchen nicht jede der vier möglichen Transitionsbeschriftungen explizit.

R$_{16}$ Dann gilt $t = \mathbf{ch}\ C.\ u$, $u \xrightarrow{c!v,A} u'$, $c \notin C$, $A \cap C = \emptyset$, $fc(v) \cap C \neq \emptyset$, $\omega = (c!v, A \cup (fc(v) \cap C))$ und $t' = \mathbf{ch}\ C \setminus fc(v).\ u'$. Mit T$_{12}$ wissen wir, daß $\vdash u : \mathit{proc}$, $\vdash c : T\ \mathit{channel}$ und $\vdash v : T$ gilt. Nach Anwendung der Induktionshypothese erhalten wir $\vdash u' : \mathit{proc}$. Dann können wir mit T$_{12}$ folgern, daß $\vdash \mathbf{ch}\ C \setminus fc(v).\ u' : \mathit{proc}$ gilt.

□

Theorem 3.8. $\sim_\beta$ ist eine Kongruenz für alle Operatoren aus $\mathcal{P}$.

Beweis für Theorem 3.8. Wir zeigen die Kongruenzeigenschaft von $\sim_\beta$ für die Operatoren von $\mathcal{P}$. Für jeden Transitionsschritt $t \xrightarrow{c!v,C} t'$ nehmen wir an, daß $C \cap fc(t) = \emptyset$ gilt (siehe Abbildung 3.3). Wir unterteilen den Beweis in zwei Abschnitte. Zuerst untersuchen wir die Fälle, für die wir das Lemma 3.9 nicht benötigen. Aufbauend auf diesem Abschnitt beweisen wir die Gültigkeit von Lemma 3.9. Nach dem Beweis des Lemmas behandeln wir die Fälle, die das Lemma verwenden.

- $\mathit{spawn}(t) \sim_\beta \mathit{spawn}(u)$ falls $t \sim_\beta u$.

  Sei $R = \{(\mathit{spawn}(t_0), \mathit{spawn}(u_0)) \mid t_0 \sim_\beta u_0\}$. Wir zeigen, daß R die Anforderungen aus Definition 3.7 erfüllt. Da R symmetrisch ist, führen wir den Beweis nur für eine Richtung der Relation; der Beweis für die andere Richtung geschieht analog.

  Wir nehmen an, daß $\mathit{spawn}(t_0) \xrightarrow{\omega} t'$ gilt und unterscheiden die folgende Fälle:

  - $t_0 \xrightarrow{c?v} t_0'$, $\omega = c?v$ und $t' = \mathit{spawn}(t_0')$, abgeleitet mit R$_8$. Aus $t_0 \sim_\beta u_0$ folgt $\forall v', v =_\beta v', \exists u_0' : u_0 \xrightarrow{c?v'} u_0'$ und $t_0' \sim_\beta u_0'$. Dann können wir mit R$_8$ folgern, daß $\mathit{spawn}(u_0) \xrightarrow{c?v'} \mathit{spawn}(u_0')$ gilt. Weiterhin gilt $(\mathit{spawn}(t_0'), \mathit{spawn}(u_0')) \in R$.
  - $t_0 \xrightarrow{\alpha} t_0'$, $\alpha = \tau$ oder $\alpha = c!v, C$, $\omega = \alpha$ und $t' = \mathit{spawn}(t_0')$. Aus $t_0 \sim_\beta u_0$ folgt $\exists \omega', \omega =_\beta \omega', \exists u_0' : u_0 \xrightarrow{\omega'} u_0'$ und $t_0' \sim_\beta u_0'$. Dann können wir mit R$_8$ folgern, daß $\mathit{spawn}(u_0) \xrightarrow{\omega'} \mathit{spawn}(u_0')$ gilt. Weiterhin gilt $(\mathit{spawn}(t_0'), \mathit{spawn}(u_0')) \in R$.
  - $\omega = \checkmark$ und $t' = \mathit{spawn}(t_0)$, abgeleitet mit R$_7$. Analog können wir mit R$_7$ folgern, daß $\mathit{spawn}(u_0) \xrightarrow{\checkmark} \mathit{spawn}(u_0)$ gilt. Weiterhin gilt $(\mathit{spawn}(t_0), \mathit{spawn}(u_0)) \in R$.

- $t_1 + u \sim_\beta t_2 + u$ falls $t_1 \sim_\beta t_2$.

  Sei $R = \{(u_1 + u, u_2 + u) \mid u_1 \sim_\beta u_2\} \cup \sim_\beta$. Wir zeigen, daß R die Anforderungen aus Definition 3.7 erfüllt. Da R symmetrisch ist, führen wir

den Beweis nur für eine Richtung der Relation; der Beweis für die andere Richtung geschieht analog.

Wir nehmen an, daß $u_1 + u \xrightarrow{\omega} u_{new}$ gilt und unterscheiden die folgenden Fälle:

- $u_1 \xrightarrow{c?v} u_1'$, $\omega = c?v$ und $u_{new} = u_1'$, abgeleitet mit R_9. Aus $u_1 \sim_\beta u_2$ folgt $\forall v', v =_\beta v', \exists u_2' : u_2 \xrightarrow{c?v'} u_2'$ und $u_1' \sim_\beta u_2'$. Daher erhalten wir mit R_9 $u_2 + u \xrightarrow{c?v'} u_2'$. Weiterhin gilt $(u_1', u_2') \in R$.
- $u_1 \xrightarrow{\omega} u_1'$, $\omega \neq c?v$ und $u_{new} = u_1'$, abgeleitet mit R_9. Aus $u_1 \sim_\beta u_2$ folgt $\exists \omega', \omega =_\beta \omega', \exists u_2' : u_2 \xrightarrow{\omega'} u_2'$ und $u_1' \sim_\beta u_2'$. Daher erhalten wir mit R_9 $u_2 + u \xrightarrow{\omega'} u_2'$. Weiterhin gilt $(u_1', u_2') \in R$.
- $u \xrightarrow{\omega} u'$ und $u_{new} = u'$, abgeleitet mit R_{10}. Ebenso können wir mit R_{10} schließen, daß $u_2 + u \xrightarrow{\omega} u'$ gilt. Weiterhin gilt $(u', u') \in R$.

- **ch** $C.\, t \sim_\beta$ **ch** $C.\, u$ falls $t \sim_\beta u$.

  Sei $R = \{(\textbf{ch}\ D.\, t_0, \textbf{ch}\ D.\, u_0) \mid t_0 \sim_\beta u_0\}$. Wir zeigen, daß R die Anforderungen aus Definition 3.7 erfüllt. Da R symmetrisch ist, führen wir den Beweis nur für eine Richtung der Relation; der Beweis für die andere Richtung geschieht analog.

  Wir nehmen an, daß **ch** $C.\, t_0 \xrightarrow{\omega} t'$ gilt und unterscheiden die folgenden Fälle:

  - $t_0 \xrightarrow{c?v} t_0'$, $\omega = c?v$, $c \notin C$, $fc(v) \cap C = \emptyset$ und $t' =$ **ch** $C.\, t_0'$, abgeleitet mit R_{15}. Aus $t_0 \sim_\beta u_0$ folgt $\forall v', v =_\beta v', \exists u_0' : u_0 \xrightarrow{c?v'} u_0'$ und $t_0' \sim_\beta u_0'$. Weiterhin wissen wir mit $v =_\beta v'$, daß $fc(v) = fc(v')$ gilt; daher können wir $fc(v') \cap C = \emptyset$ folgern. Durch Anwendung von R_{15} erhalten wir **ch** u_0. $\xrightarrow{c?v'}$ **ch** $C.\, u_0'$. Weiterhin gilt $(\textbf{ch}\ C.\, t_0', \textbf{ch}\ C.\, u_0') \in R$.
  - $t_0 \xrightarrow{\omega} t_0'$, $\omega \neq c?v$, $fc(\omega) \cap C = \emptyset$ und $t' =$ **ch** $C.\, t_0'$, abgeleitet mit R_{15}. Aus $t_0 \sim_\beta u_0$ folgt $\exists \omega', \omega =_\beta \omega', \exists u_0' : u_0 \xrightarrow{\omega'} u_0'$ und $t_0' \sim_\beta u_0'$. Weiterhin wissen wir mit $\omega =_\beta \omega'$, daß $fc(\omega) = fc(\omega')$ gilt; daher können wir $fc(\omega') \cap C = \emptyset$ folgern. Durch Anwendung von R_{15} erhalten wir **ch** $C.\, u_0 \xrightarrow{\omega'}$ **ch** $C.\, u_0'$. Weiterhin gilt $(\textbf{ch}\ C.\, t_0', \textbf{ch}\ C.\, u_0') \in R$.
  - $t_0 \xrightarrow{c!v,A} t_0'$, $A \cap C = \emptyset$, $fc(v) \cap C \neq \emptyset$, $\omega = (c!v, A \cup (C \cap fc(v)))$ und $t' =$ **ch** $C \setminus fc(v).t_0'$, abgeleitet mit R_{16}. Aus $t_0 \sim_\beta u_0$ folgt $\exists v', v =_\beta v', \exists u_0' :$ $u_0 \xrightarrow{c!v',A} u_0'$ und $t_0' \sim_\beta u_0'$. Weiterhin wissen wir mit $v =_\beta v'$, daß $fc(v) = fc(v')$ gilt; daher können wir $fc(v') \cap C = fc(v) \cap C \neq \emptyset$ folgern. Dann erhalten wir mit R_{16} **ch** $C.\, u_0 \xrightarrow{c!v', A \cup (fc(v) \cap C)}$ **ch** $C \setminus fc(v).\, u_0'$. Weiterhin gilt $(\textbf{ch}\ C \setminus fc(v).\, t_0', \textbf{ch}\ C \setminus fc(v).\, u_0') \in R$.

- $t_1; t_2 \sim_\beta u_1; u_2$ falls $t_1 \sim_\beta u_1$ und $t_2 \sim_\beta u_2$.

Sei $R = \{(\mathbf{ch}\ D.\, t_1; t_2, \mathbf{ch}\ D.\, u_1; u_2) \mid t_1 \sim_\beta u_1, t_2 \sim_\beta u_2\}$. Wir zeigen, daß R die Anforderungen aus Definition 3.7 erfüllt. Da R symmetrisch ist, führen wir den Beweis nur für eine Richtung der Relation; der Beweis für die andere Richtung geschieht analog.

Wir führen den Beweis für $D = \emptyset$. Für $D \neq \emptyset$ können Beweistechniken analog dem Kongruenzbeweis für Kanalerzeugung angewendet werden. Wir nehmen an, daß $t_1; t_2 \xrightarrow{\omega} t_{new}$ gilt und unterscheiden die folgenden Fälle:

- $t_1 \xrightarrow{c?v} t_1'$, $\omega = c?v$ und $t_{new} = t_1'; t_2$, abgeleitet mit R_{12}. Aus $t_1 \sim_\beta u_1$ folgt $\forall v', v =_\beta v', \exists u_1' : u_1 \xrightarrow{c?v'} u_1'$ und $t_1' \sim_\beta u_1'$. Dann können wir mit R_{12} folgern, daß $u_1; u_2 \xrightarrow{c?v'} u_1'; u_2$ gilt. Weiterhin gilt $(t_1'; t_2, u_1'; u_2) \in R$.
- $t_1 \xrightarrow{\alpha} t_1'$, $\alpha \neq c?v$, $\omega = \alpha$ und $t_{new} = t_1; t_2$, abgeleitet mit R_{12}. Aus $t_1 \sim_\beta u_1$ folgt $\exists \omega', \omega =_\beta \omega', \exists u_1' : u_1 \xrightarrow{\omega'} u_1'$ und $t_1' \sim_\beta u_1'$. Dann können wir mit R_{12} folgern, daß $u_1; u_2 \xrightarrow{\omega'} u_1'; u_2$ gilt. Weiterhin gilt $(t_1'; t_2, u_1'; u_2) \in R$.
- $t_1 \xrightarrow{\checkmark} t_1'$, $t_2 \xrightarrow{c?v} t_2'$, $\omega = c?v$ und $t_{new} = t_1'; t_2'$, abgeleitet mit R_{13}. Aus $t_1 \sim_\beta u_1$ folgt $\exists u_1' : u_1 \xrightarrow{\checkmark} u_1'$ und $t_1' \sim_\beta u_1'$. Aus $t_2 \sim_\beta u_2$ folgt $\forall v', v =_\beta v', \exists u_2' : u_2 \xrightarrow{c?v'} u_2'$ und $t_2' \sim_\beta u_2'$. Daher können wir mit R_{13} schließen, daß $u_1; u_2 \xrightarrow{c?v'} u_1'; u_2'$ gilt. Weiterhin gilt $(t_1'; t_2', u_1'; u_2') \in R$.
- $t_1 \xrightarrow{\checkmark} t_1'$, $t_2 \xrightarrow{\omega} t_2'$, $\omega \neq c?v$ und $t_{new} = t_1'; t_2'$, abgeleitet mit R_{13}. Aus $t_1 \sim_\beta u_1$ folgt $\exists u_1' : u_1 \xrightarrow{\checkmark} u_1'$ und $t_1' \sim_\beta u_1'$. Aus $t_2 \sim_\beta u_2$ folgt $\exists \omega', \omega =_\beta \omega', \exists u_2' : u_2 \xrightarrow{\omega'} u_2'$ und $t_2' \sim_\beta u_2'$. Daher können wir mit R_{13} schließen, daß $u_1; u_2 \xrightarrow{\omega'} u_1'; u_2'$ gilt. Weiterhin gilt $(t_1'; t_2', u_1'; u_2') \in R$.
- $t_1 \xrightarrow{\checkmark} t_1'$, $t_1' \xrightarrow{c?v} t_1''$, $t_2 \xrightarrow{c!v,C} t_2'$, $\omega = \tau$ und $t_{new} = \mathbf{ch}\ C.\, t_1''; t_2'$, abgeleitet mit R_{14}. Aus $t_2 \sim_\beta u_2$ folgt $\exists v', v =_\beta v', \exists u_2' : u_2 \xrightarrow{c!v',C} u_2'$ und $t_2' \sim_\beta u_2'$. Aus $t_1 \sim_\beta u_1$ folgt $\exists u_1' : u_1 \xrightarrow{\checkmark} u_1'$, $t_1' \sim_\beta u_1'$ und $\forall v'', v =_\beta v'', \exists u_1'' : u_1' \xrightarrow{c?v''} u_1''$, $t_1'' \sim_\beta u_1''$. Dann wissen wir mit $v =_\beta v'$, daß $\exists u_v : u_1' \xrightarrow{c?v'} u_v$ und $t_1'' \sim_\beta u_v$. Daher können wir mit R_{14} folgern, daß $u_1; u_2 \xrightarrow{\tau} \mathbf{ch}\ C.u_v; u_2'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.t_1''; t_2', \mathbf{ch}\ C.u_v; u_2') \in R$.
- $t_1 \xrightarrow{\checkmark} t_1'$, $t_1' \xrightarrow{c!v,C} t_1''$, $t_2 \xrightarrow{c?v} t_2'$, $\omega = \tau$ und $t_{new} = \mathbf{ch}\ C.\, t_1''; t_2'$, abgeleitet mit R_{14}. Aus $t_1 \sim_\beta u_1$ folgt $\exists u_1' : u_1 \xrightarrow{\checkmark} u_1'$, $t_1' \sim_\beta u_1'$ and $\exists v', v =_\beta v', \exists u_1'' : u_1' \xrightarrow{c!v',C} u_1''$, $t_1'' \sim_\beta u_1''$. Aus $t_2 \sim_\beta u_2$ folgt $\forall v'', v =_\beta v'', \exists u_2' : u_2 \xrightarrow{c?v''} u_2'$ and $t_2' \sim_\beta u_2'$. Dann wissen wir mit $v =_\beta v'$, daß $\exists u_v : u_2 \xrightarrow{c?v'} u_v$ und $t_2' \sim_\beta u_v$. Daher können wir mit R_{14} folgern, daß $u_1; u_2 \xrightarrow{\tau} \mathbf{ch}\ C.u_1''; u_v$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.t_1''; t_2', \mathbf{ch}\ C.u_1'', u_v) \in R$.

Die Beweise für die übrigen Operatoren verwenden Lemma 3.9, daher zeigen wir zunächst die Gültigkeit dieses Lemmas. Dieser Beweis verwendet Kongruenzresultate aus den bisher bewiesenen Fällen von Theorem 3.8.

Lemma 3.9. Wenn $\sigma =_\beta \sigma'$, dann gilt $t\sigma \sim_\beta t\sigma'$.

Beweis für Lemma 3.9. Wir führen eine Induktion über die Termstruktur von t. Sei $R = \{(t\sigma_1, t\sigma_1') \mid t \in \mathcal{P}, \sigma =_\beta \sigma'\} \cup \sim_\beta$. Wir zeigen, daß R die Anforderungen aus Definition 3.7 erfüllt. Für jeden Transitionsschritt $t \xrightarrow{c!v,C} t'$ nehmen wir an, daß $C \cap fc(t) = \emptyset$ gilt (siehe Abbildung 3.3). Da R symmetrisch ist, führen wir den Beweis nur für eine Richtung der Relation; der Beweis für die andere Richtung geschieht analog.

- Die Fälle $t = \mathbf{0}$, $t = \mathbf{1}$ und $t = \tau$ sind trivial.
- $t = [E]$. Aus $\sigma =_\beta \sigma'$ folgt $\exists E', E'', E' =_\beta E'' : t\sigma = [E']$ und $t\sigma' = [E'']$. Mit $E' =_\beta E''$ wissen wir, daß $[\![E']\!] = [\![E'']\!]$ gilt; daher folgt $[E''] \xrightarrow{\tau} \mathbf{1}$ aus $[E'] \xrightarrow{\tau} \mathbf{1}$, abgeleitet mit R_3. Weiterhin gilt $(\mathbf{1}, \mathbf{1}) \in R$.
- $t = E!F$. Aus $\sigma =_\beta \sigma'$ folgt $\exists E', E'', E' =_\beta E'', \exists F', F'', F' =_\beta F'' : t\sigma = E'!F'$ und $t\sigma' = E''!F''$. Für Kanäle entspricht β-Äquivalenz der Identität, daher wissen wir, daß $\exists c : [\![E']\!] = c = [\![E'']\!]$ gilt. Aus $F' =_\beta F''$ folgt $\exists v, v', v =_\beta v' : [\![F']\!] = v$ und $[\![F'']\!] = v'$. Dann erhalten wir mit R_4 $t\sigma \xrightarrow{c!v} \mathbf{1}$, $t\sigma' \xrightarrow{c!v'} \mathbf{1}$ und $c!v =_\beta c!v'$. Weiterhin gilt $(\mathbf{1}, \mathbf{1}) \in R$.
- $t = E?x; u$. Aus $\sigma =_\beta \sigma'$ folgt $\exists E', E'', E' =_\beta E''$, $t\sigma = E'?x; u\sigma$ und $t\sigma' = E''?x; u\sigma'$. Für Kanäle entspricht β-Äquivalenz der Identität, daher wissen wir, daß $\exists c : [\![E']\!] = c = [\![E'']\!]$ gilt. Dann können wir mit R_5 schließen, daß $E'?x; u\sigma \xrightarrow{c?v} u\sigma\langle v/x\rangle$ und $\forall v', v =_\beta v' : E''?x; u\sigma' \xrightarrow{c?v'} u\sigma'\langle v'/x\rangle$ gilt. Weiterhin gilt $(u\sigma\langle v/x\rangle, u\sigma'\langle v'/x\rangle) \in R$.
- $t = \{\vec{x} \leftarrow \vec{E}\}; u$. Aus $\sigma =_\beta \sigma'$ folgt $\exists \vec{E}', \vec{E}'', \vec{E}' =_\beta \vec{E}'' : t\sigma = \{\vec{x} \leftarrow \vec{E}'\}; u\sigma$ und $t\sigma' = \{\vec{x} \leftarrow \vec{E}''\}; u\sigma'$. Sei $[\![\vec{E}']\!] = \vec{v}'$ und sei $[\![\vec{E}'']\!] = \vec{v}''$. Aus $\vec{E}' =_\beta \vec{E}''$ folgt $\vec{v}' =_\beta \vec{v}''$. Wir nehmen $t\sigma \xrightarrow{\omega} u'$ an. Dann gilt $u\sigma\langle \vec{v}'/\vec{x}\rangle \xrightarrow{\omega} u'$. Nach Anwendung der Induktionshypothese erhalten wir $u\sigma\langle \vec{v}'/\vec{x}\rangle \sim_\beta u\sigma'\langle \vec{v}''/\vec{x}\rangle$. Daher können wir, mit einer adäquaten Unterscheidung für ω, daß $\exists u'' : u\sigma'\langle \vec{v}''/\vec{x}\rangle \xrightarrow{\omega'} u''$ und $u' \sim_\beta u''$ gilt. Dann können wir mit R_6 folgern, daß $t\sigma' \xrightarrow{\omega} u''$ gilt. Weiterhin gilt $(u', u'') \in R$.
- $t = \mathit{spawn}(u)$. Dann gilt $t\sigma = \mathit{spawn}(u\sigma)$ und $t\sigma' = \mathit{spawn}(u\sigma')$. Nach Induktionshypothese wissen wir $u\sigma \sim_\beta u\sigma'$. Wir nehmen $t\sigma \xrightarrow{\omega} t'$ an und unterscheiden die folgenden Fälle:
  - $u\sigma \xrightarrow{c?v} u'$, $\omega = c?v$ und $t' = \mathit{spawn}(u')$, abgeleitet mit R_8. Aus $u\sigma \sim_\beta u\sigma'$ folgt $\forall v', v =_\beta v', \exists u'' : u\sigma' \xrightarrow{c?v'} u''$ und $u' \sim_\beta u''$. Dann

können wir mit R_8 folgern, daß $spawn(u\sigma') \xrightarrow{c?v'} spawn(u'')$ gilt. Aus der oben gezeigten Kongruenz von $\sim_\beta$ bzgl. *spawn* können wir folgern, daß $(spawn(u'), spawn(u'')) \in R$ gilt.

- $u\sigma \xrightarrow{\alpha} u'$, $\alpha \neq c?v$, $\omega = \alpha$ und $t' = spawn(u')$, abgeleitet mit R_8. Aus $u\sigma \sim_\beta u\sigma'$ folgt $\exists \alpha', \alpha =_\beta \alpha', \exists u'' : u\sigma' \xrightarrow{\alpha'} u''$ und $u' \sim_\beta u''$. Dann können wir mit R_8 folgern, daß $spawn(u\sigma') \xrightarrow{\alpha'} spawn(u'')$ gilt. Weiterhin gilt $(spawn(u'), spawn(u'')) \in R$.
- $\omega = \checkmark$ und $t' = spawn(u\sigma)$, abgeleitet mit R_7. Ebenso können wir mit R_7 ableiten, daß $spawn(u\sigma') \xrightarrow{\checkmark} spawn(u\sigma')$ gilt. Weiterhin gilt $(spawn(u\sigma), spawn(u\sigma')) \in R$.

- $t = t_1 + t_2$. Dann gilt $t\sigma = (t_1 + t_2)\sigma = t_1\sigma + t_2\sigma$ und $t\sigma' = (t_1 + t_2)\sigma' = t_1\sigma' + t_2\sigma'$. Nach Induktionshypothese wissen wir, daß $t_1\sigma \sim_\beta t_1\sigma'$ und $t_2\sigma \sim_\beta t_2\sigma'$ gilt. Wir nehmen $t\sigma \xrightarrow{\omega} t'$ an und unterscheiden die folgenden Fälle:

  - $t_1\sigma \xrightarrow{c?v} t_1'$, $\omega = c?v$ und $t' = t_1'$, abgeleitet mit R_9. Aus $t_1\sigma \sim_\beta t_1\sigma'$ folgt $\forall v', v =_\beta v', \exists t_1'' : t_1\sigma' \xrightarrow{c?v'} t_1''$ und $t_1' \sim_\beta t_1''$. Daher können wir mit R_9 folgern, daß $t_1\sigma' + t_2\sigma' \xrightarrow{c?v'} t_1''$ gilt. Weiterhin gilt $(t_1', t_1'') \in R$.
  - $t_1\sigma \xrightarrow{\omega} t_1'$, $\omega \neq c?v$ und $t' = t_1'$, abgeleitet mit R_9. Aus $t_1\sigma \sim_\beta t_1\sigma'$ folgt $\exists \omega', \omega =_\beta \omega', \exists t_1'' : t_1\sigma' \xrightarrow{\omega'} t_1''$ und $t_1' \sim_\beta t_1''$. Daher können wir mit R_9 folern, daß $t_1\sigma' + t_2\sigma' \xrightarrow{\omega'} t_1''$ gilt. Weiterhin gilt $(t_1', t_1'') \in R$.

  Der Beweis für den Fall $t_2\sigma \xrightarrow{\omega} t_2'$ ist analog.

- $t = \mathbf{ch}\ C.\ u$. Für $C \cap \mathrm{rng}(\sigma) \neq \emptyset$ oder $C \cap \mathrm{rng}(\sigma') \neq \emptyset$ wenden wir α-Konversion auf t an. Es gilt $t\sigma = \mathbf{ch}\ C.\ u\sigma$ und $t\sigma' = \mathbf{ch}\ C.\ u\sigma'$. Nach Anwedung der Induktionshypothese wissen wir, daß $u\sigma \sim_\beta u\sigma'$ gilt. Wir nehmen $t\sigma \xrightarrow{\omega} t'$ an und unterscheiden die folgenden Fälle:

  - $u\sigma \xrightarrow{c?v} u'$, $c \notin C$, $fc(v) \cap C = \emptyset$, $\omega = c?v$ und $t' = \mathbf{ch}\ C.\ u'$, abgeleitet mit R_{15}. Aus $u\sigma \sim_\beta u\sigma'$ folgt $\forall v', v =_\beta v', \exists u'' : u\sigma' \xrightarrow{c?v'} u''$ und $u' \sim_\beta u''$. Aus $v =_\beta v'$ folgt $fc(v) = fc(v')$. Dann können wir mit R_{15} schließen, daß $\mathbf{ch}\ C.\ u\sigma' \xrightarrow{c?v'} \mathbf{ch}\ C.\ u''$ gilt. Weiterhin wissen wir aus dem obigen Beweis der Kongruenz von $\sim_\beta$ bzgl. Restriktion, daß $(\mathbf{ch}\ C.\ u', \mathbf{ch}\ C.\ u'') \in R$ gilt.
  - $u\sigma \xrightarrow{\omega} u'$, $\omega \neq c?v$, $fc(\omega) \cap C = \emptyset$ und $t' = \mathbf{ch}\ C.u'$, abgeleitet mit R_{15}. Aus $u\sigma =_\beta u\sigma'$ folgt $\exists \omega', \omega' =_\beta \omega, \exists u'' : u\sigma' \xrightarrow{\omega'} u''$ und $u' \sim_\beta u''$. Aus $\omega =_\beta \omega'$ folgt $fc(\omega) = fc(\omega')$. Dann können wir mit R_{15} schließen, daß $\mathbf{ch}\ C.\ u\sigma' \xrightarrow{\omega'} \mathbf{ch}\ C.\ u''$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\ u', \mathbf{ch}\ C.\ u'') \in R$.

– $u\sigma \xrightarrow{c!v,A} u'$, $c \notin C$, $A \cap C = \emptyset$, $fc(v) \cap C \neq \emptyset$, $\omega = A \cup (C \cap fc(v))$ und $t' = \mathbf{ch}\ C \setminus fc(v).\, u'$, abgeleitet mit R_{16}. Aus $u\sigma \sim_\beta u\sigma'$ folgt $\exists v', v =_\beta v', \exists u'' : u\sigma' \xrightarrow{c!v',A} u''$ und $u' \sim_\beta u''$. Aus $v =_\beta v'$ folgt $fc(v) = fc(v')$. Dann können wir mit R_{16} schließen, daß $\mathbf{ch}\ C.\, u\sigma' \xrightarrow{c!v',A\cup(C\cap fc(v))} \mathbf{ch}\ C \setminus fc(v).\, u''$ gilt. Weiterhin gilt $(\mathbf{ch}\ C \setminus fc(v).\, u', \mathbf{ch}\ C \setminus fc(v).\, u'') \in R$.

- $t = P(\vec{E})$. Sei $\Theta : P(\vec{x} : \vec{T}) \mapsto u$. Aus $\sigma =_\beta \sigma'$ folgt $\exists \vec{E'}, \vec{E''}, \vec{E'} =_\beta \vec{E''} : t\sigma = P(\vec{E'})$ und $t\sigma' = P(\vec{E''})$. Sei $\vec{v'} = [\![\vec{E'}]\!]$ und sei $\vec{v''} = [\![\vec{E''}]\!]$. Aus $\vec{E'} =_\beta \vec{E''}$ folgt $\vec{v'} =_\beta \vec{v''}$. Mit R_{11} können wir folgern, daß $P(\vec{E'}) \xrightarrow{\tau} u\langle \vec{v'}/\vec{x}\rangle$ und $P(\vec{E''}) \xrightarrow{\tau} u\langle \vec{v''}/\vec{x}\rangle$ gilt. Weiterhin gilt $(u\langle \vec{v'}/\vec{x}\rangle, u\langle \vec{v''}/\vec{x}\rangle) \in R$.

- $t = t_1; t_2$. Dann gilt $t\sigma = (t_1; t_2)\sigma = t_1\sigma; t_2\sigma$ und $t\sigma' = (t_1; t_2)\sigma' = t_1\sigma'; t_2\sigma'$. Mit der Induktionshypothese wissen wir, daß $t_1\sigma \sim_\beta t_1\sigma'$ und $t_2\sigma \sim_\beta t_2\sigma'$ gilt. Wir nehmen $t\sigma \xrightarrow{\omega} t'$ an und unterscheiden die folgenden Fälle:

  – $t_1\sigma \xrightarrow{c?v} t_1'$, $\omega = c?v$ und $t' = t_1'; t_2\sigma$, abgeleitet mit R_{12}. Aus $t_1\sigma \sim_\beta t_1\sigma'$ folgt $\forall v', v =_\beta v', \exists t_1'' : t_1\sigma' \xrightarrow{c?v'} t_1''$ und $t_1' \sim_\beta t_1''$. Dann können wir mit R_{12} folgern, daß $t_1\sigma'; t_2\sigma' \xrightarrow{c?v'} t_1''; t_2\sigma$ gilt. Weiterhin wissen wir aus der oben gezeigten Kongruenzeigenschaft von $\sim_\beta$ bzgl. sequentieller Komposition, daß $(t_1'; t_2\sigma, t_1''; t_2\sigma) \in R$ gilt.

  – $t_1\sigma \xrightarrow{\alpha} t_1'$, $\alpha \neq c?v$, $\omega = \alpha$ und $t' = t_1'; t_2\sigma$, abgeleitet mit R_{12}. Aus $t_1\sigma \sim_\beta t_1\sigma'$ folgt $\exists \alpha', \alpha =_\beta \alpha', \exists t_1'' : t_1\sigma' \xrightarrow{\alpha'} t_1''$ und $t_1' \sim_\beta t_1''$. Dann können wir mit R_{12} schließen, daß $t_1\sigma'; t_2\sigma' \xrightarrow{\alpha'} t_1''; t_2\sigma$ gilt. Weiterhin gilt $(t_1'; t_2\sigma, t_1''; t_2\sigma) \in R$.

  – $t_1\sigma \xrightarrow{\checkmark} t_1'$, $t_2\sigma \xrightarrow{c?v} t_2'$, $\omega = c?v$ und $t' = t_1'; t_2'$, abgeleitet mit R_{13}. Aus $t_1\sigma \sim_\beta t_1\sigma'$ folgt $\exists t_1'' : t_1\sigma' \xrightarrow{\checkmark} t_1''$ und $t_1' \sim_\beta t_1''$. Aus $t_2\sigma \sim_\beta t_2\sigma'$ folgt $\forall v', v =_\beta v', \exists t_2'' : t_2\sigma' \xrightarrow{c?v'} t_2''$ und $t_2' \sim_\beta t_2''$. Dann können wir mit R_{13} schließen, daß $t_1\sigma'; t_2\sigma' \xrightarrow{c?v'} t_1''; t_2''$. Weiterhin gilt $(t_1'; t_2', t_1''; t_2'') \in R$.

  – $t_1\sigma \xrightarrow{\checkmark} t_1'$, $t_2\sigma \xrightarrow{\omega} t_2'$, $\omega \neq c?v$ und $t' = t_1'; t_2'$, abgeleitet mit R_{13}. Aus $t_1\sigma \sim_\beta t_1\sigma'$ folgt $\exists t_1'' : t_1\sigma' \xrightarrow{\checkmark} t_1''$ und $t_1' \sim_\beta t_1''$. Aus $t_2\sigma \sim_\beta t_2\sigma'$ folgt $\exists \omega', \omega =_\beta \omega', \exists t_2'' : t_2\sigma' \xrightarrow{\omega'} t_2''$ und $t_2' \sim_\beta t_2''$. Dann können wir mit R_{13} schließen, daß $t_1\sigma'; t_2\sigma' \xrightarrow{\omega'} t_1''; t_2''$. Weiterhin gilt $(t_1'; t_2', t_1''; t_2'') \in R$.

  – $t_1\sigma \xrightarrow{\checkmark} t_1'$, $t_1' \xrightarrow{c?v} t_1''$, $t_2\sigma \xrightarrow{c!v,C} t_2'$, $\omega = \tau$ und $t' = \mathbf{ch}\ C.\, t_1''; t_2'$, abgeleitet mit R_{14}. Aus $t_2\sigma \sim_\beta t_2\sigma'$ folgt $\exists v', v =_\beta v', \exists t_2'' : t_2\sigma' \xrightarrow{c!v',C} t_2''$ und $t_2' \sim_\beta t_2''$. Aus $t_1\sigma \sim_\beta t_1\sigma'$ folgt $\exists t_3' : t_1\sigma' \xrightarrow{\checkmark} t_3'$, $t_1' \sim_\beta t_3'$ und $\forall v'', v =_\beta v'', \exists t_3'' : t_3' \xrightarrow{c?v''} t_3''$, $t_1'' \sim_\beta t_3''$. Dann wissen wir mit $v' =_\beta v''$, daß $\exists t_v : t_3' \xrightarrow{c?v'} t_v$ und $t_1'' \sim_\beta t_v$ gilt. Daher können

wir mit R_{14} folgern, daß $t_1\sigma'; t_2\sigma' \xrightarrow{\tau} \mathbf{ch}\ C.\ t_v; t_2''$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\ t_1''; t_2', \mathbf{ch}\ C.\ t_v; t_2'') \in R$.

– $t_1\sigma \xrightarrow{\checkmark} t_1'$, $t_1' \xrightarrow{c!v,C} t_1''$, $t_2\sigma \xrightarrow{c?v} t_2'$, $\omega = \tau$ und $t' = \mathbf{ch}\ C.\ t_1''; t_2'$, abgeleitet mit R_{14}. Aus $t_1\sigma \sim_\beta t_1\sigma'$ folgt $\exists t_3' : t_1\sigma' \xrightarrow{\checkmark} t_3'$, $t_1' \sim_\beta t_3'$ und $\exists v', v =_\beta v', \exists t_3'' : t_3' \xrightarrow{c!v',C} t_3''$, $t_1'' \sim_\beta t_3''$. Aus $t_2\sigma \sim_\beta t_2\sigma'$ folgt $\forall v'', v =_\beta v'', \exists t_2'' : t_2\sigma' \xrightarrow{c?v''} t_2''$ und $t_2' \sim_\beta t_2''$. Dann wissen wir mit $v' =_\beta v''$, daß $\exists t_v : t_2\sigma' \xrightarrow{c?v'} t_v$ und $t_2' \sim_\beta t_v$ gilt. Daher können wir mit R_{14} folgern, daß $t_1\sigma'; t_2\sigma' \xrightarrow{\tau} \mathbf{ch}\ C.\ t_3''; t_v$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\ t_1''; t_2', \mathbf{ch}\ C.\ t_3''; t_v) \in R$.

□

(Fortführung des Beweises für Theorem 3.8)
Nun zeigen wir die Kongruenzeigenschaft von $\sim_\beta$ bzgl. der übrigen Operatoren aus $\mathcal{P}$. Die Beweise verwenden Lemma 3.9.

- $\{\vec{x} \leftarrow \vec{E}\}; t \sim_\beta \{\vec{x} \leftarrow \vec{E}'\}; u$ falls $t \sim_\beta u$ und $\vec{E} =_\beta \vec{E}'$.

  Sei $R = \{(\{\vec{x} \leftarrow \vec{E}\}; t_0, \{\vec{x} \leftarrow \vec{E}'\}; u_0) \mid t_0 \sim_\beta u_0, \vec{E} =_\beta \vec{E}'\} \cup \sim_\beta$. Wir zeigen, daß R die in Definition 3.7 geforderten Eigenschaften erfüllt. Wegen der Symmetrie von R führen wir den Beweis nur für eine Richtung.

  Wir nehmen $\{\vec{x} \leftarrow \vec{E}\}; t_0 \xrightarrow{\omega} t'$ an und unterscheiden die folgenden Fälle:

  – $[\![\vec{E}]\!] = \vec{v}$, $t_0\langle\vec{v}/\vec{x}\rangle \xrightarrow{c?v} t_0'$ und $t' = t_0'$, abgeleitet mit R_6. Sei $\vec{v}' = [\![\vec{E}']\!]$. Mit $t_0 \sim_\beta u_0$ und Lemma 3.9 können wir schließen, daß $t_0\langle\vec{v}/\vec{x}\rangle \sim_\beta u_0\langle\vec{v}'/\vec{x}\rangle$ gilt. Dann wissen wir auch, daß $\forall v', v' =_\beta v, \exists u_0' : u_0\langle\vec{v}'/\vec{x}\rangle \xrightarrow{c?v'} t_0'$ und $t_0' \sim_\beta u_0'$ gilt. Daher können wir mit R_6 schließen, daß $\{\vec{x} \leftarrow \vec{E}'\}; u_0 \xrightarrow{c?v'} u_0'$ gilt. Weiterhin gilt $(t_0', u_0') \in R$.

  – $[\![\vec{E}]\!] = \vec{v}$, $t_0\langle\vec{v}/\vec{x}\rangle \xrightarrow{\omega} t_0'$, $\omega \neq c?v$ und $t' = t_0'$, abgeleitet mit R_6. Sei $\vec{v}' = [\![\vec{E}']\!]$. Mit $t_0 \sim_\beta u_0$ und Lemma 3.9 können wir schließen, daß $t_0\langle\vec{v}/\vec{x}\rangle \sim_\beta u_0\langle\vec{v}'/\vec{x}\rangle$. Dann wissen wir, daß $\exists\omega', \omega =_\beta \omega', \exists u_0' : u_0\langle\vec{v}'/\vec{x}\rangle \xrightarrow{\omega'} u_0'$ und $t_0' \sim_\beta u_0'$. Daher können wir mit R_6 schließen, daß $\{\vec{x} \leftarrow \vec{E}'\}; u_0 \xrightarrow{\omega'} u_0'$ gilt. Weiterhin gilt $(t_0', u_0') \in R$.

- $E?x; t \sim_\beta E'?x; u$ falls $t \sim_\beta u$ und $E =_\beta E'$.

  Sei $R = \{(E?x; t_0, E'?x; u_0) \mid t_0 \sim_\beta u_0, E =_\beta E'\} \cup \sim_\beta$. Wir zeigen, daß R die Anforderungen aus Definition 3.7 erfüllt. Da R symmetrisch ist, führen wir den Beweis nur für eine Richtung.

  Wir nehmen an, daß $E?x; t_0 \xrightarrow{\omega} t'$ gilt. Für Kanäle entspricht β-Äquivalenz der Identität, daher gilt $\exists c : [\![E]\!] = c = [\![E']\!]$. Dann wissen wir mit R_5, daß $\omega = c?v$ und $t' = t_0\langle v/x\rangle$ gilt. Sei $v' =_\beta v$ beliebig. Dann können

wir mit R_5 folgern, daß $E'?x; u_0 \xrightarrow{c?v'} u_0\langle v'/x\rangle$ gilt. Aus $t_0 \sim_\beta u_0$ folgt $\forall\sigma : t_0\sigma \sim_\beta u_0\sigma$. Daher können wir mit $v =_\beta v'$ und Lemma 3.9 schließen, daß $t_0\langle v/x\rangle \sim_\beta u_0\langle v'/x\rangle$ gilt. Weiterhin gilt $(t_0\langle v/x\rangle, u_0\langle v'/x\rangle) \in R$.

□

Theorem 3.10. Die axiomatische Theorie $\mathcal{AX}_{\sim_\beta}$ ist korrekt bezüglich $\sim_\beta$: Für $\mathcal{AX}_{\sim_\beta} \vdash t = u$ gilt $t \sim_\beta u$.

Beweis für Theorem 3.10. Für jedes Axiom aus Abbildung 3.4 definieren wir eine Relation und zeigen, daß sie die geforderten Eigenschaften einer Bisimulationsrelation aus Definition 3.7 erfüllt. Für jeden Transitionsschritt $t \xrightarrow{c!v,C} t'$ nehmen wir, wie in Abbildung 3.3 gefordert, an, daß $C \cap fc(t) = \emptyset$ gilt.

- Axiom (3.1): $[false] = \mathbf{0}$.

  Sei $R = \{([false], \mathbf{0})\}$. Mit R_3 wissen wir, daß $[false] \not\xrightarrow{}$ gilt. Weiterhin gilt $\mathbf{0} \not\xrightarrow{}$. Daher erfüllt R die Anforderungen aus Definition 3.7.

- Axiom (3.2): $[true] = \tau$.

  Sei $R = \{([true], \tau)\} \cup \{(t, t) \mid t \in \mathcal{P}\}$. Mit R_3 wissen wir $[true] \xrightarrow{\tau} \mathbf{1}$. Ebenso wissen wir mit R_2, daß $\tau \xrightarrow{\tau} \mathbf{1}$ gilt. Weiterhin gilt $(\mathbf{1}, \mathbf{1}) \in R$ und somit erfüllt R die Bedingungen aus Definition 3.7.

- Axiom (3.3): $\mathbf{1}; t = t$.

  Sei $R = \{(\mathbf{1}; u, u) \mid u \in \mathcal{P}\}$. Wir zeigen, daß R die in Definition 3.7 geforderten Eigenschaften besitzt. Hierzu unterscheiden wir die folgenden Fälle:

  - Wir nehmen $\mathbf{1}; u \xrightarrow{\omega} u_{new}$ an. Dann wissen wir mit R_1 und R_{13}, daß $u \xrightarrow{\omega} u'$ und $u_{new} = \mathbf{1}; u'$ gilt. Weiterhin gilt $(\mathbf{1}; u', u') \in R$.
  - Wir nehmen $u \xrightarrow{\omega} u'$ an. Dann wissen wir mit R_1 und R_{13}, daß $\mathbf{1}; u \xrightarrow{\omega} \mathbf{1}; u'$ gilt. Weiterhin gilt $(\mathbf{1}; u', u') \in R$.

- Axiom (3.4): $t; \mathbf{1} = t$.

  Sei $R = \{(u; \mathbf{1}, u) \mid u \in \mathcal{P}\}$. Wir zeigen, daß R die in Definition 3.7 gestellten Anforderungen erfüllt. Wir unterscheiden die folgenden Fälle:

  - Wir nehmen $u; \mathbf{1} \xrightarrow{\omega} u_{new}$ an und unterscheiden die folgenden Fälle:
    - $u \xrightarrow{\alpha} u'$, $\omega = \alpha$ und $u_{new} = u'; \mathbf{1}$, abgeleitet mit R_{12}. Weiterhin gilt $(u'; \mathbf{1}, u') \in R$.
    - $u \xrightarrow{\checkmark} u'$, $\omega = \checkmark$ und $u_{new} = u'; \mathbf{1}$, abgeleitet mit R_1, R_{13}. Weiterhin gilt $(u'; \mathbf{1}, u') \in R$.
  - Wir nehmen $u \xrightarrow{\omega} u'$ an und unterscheiden die folgenden Fälle:

  - $\omega = \alpha$. Dann gilt $u; \mathbf{1} \xrightarrow{\alpha} u'; \mathbf{1}$, abgeleitet mit R_{12}. Weiterhin gilt $(u'; \mathbf{1}, u') \in R$.
  - $\omega = \checkmark$. Dann gilt $u; \mathbf{1} \xrightarrow{\checkmark} u'; \mathbf{1}$, abgeleitet mit R_1, R_{13}. Weiterhin gilt $(u'; \mathbf{1}, u') \in R$.

- Axiom (3.5): $\mathbf{0}; t = \mathbf{0}$.

  Sei $R = \{(\mathbf{0}; u, \mathbf{0}) \mid u \in \mathcal{P}\}$. Mit $\mathbf{0} \not\xrightarrow{}$ und R_{12}, R_{13} wissen wir $\mathbf{0}; u \not\xrightarrow{}$. Daher erfüllt R die Anforderungen aus Definition 3.7.

- Axiom (3.6): $t_1; (t_2; t_3) = (t_1; t_2); t_3$. Sei

$$\begin{aligned}
R_1 &= \{(u_1; (u_2; u_3), (u_1; u_2); u_3) \mid u_1, u_2, u_3 \in \mathcal{P}\} \\
R_2 &= \{(\mathbf{ch}\ C.\ u_1; (u_2; u_3), (\mathbf{ch}\ C.\ u_1; u_2); u_3) \mid \\
&\qquad u_1, u_2, u_3 \in \mathcal{P}, C \cap fc(u_3) = \emptyset\} \\
R_3 &= \{(u_1; (\mathbf{ch}\ C.\ u_2; u_3), \mathbf{ch}\ C.\ (u_1; u_2); u_3) \mid \\
&\qquad u_1, u_2, u_3 \in \mathcal{P}, C \cap fc(u_1) = \emptyset\} \\
R_4 &= \{(\mathbf{ch}\ C.\ u_1; (u_2; u_3), \mathbf{ch}\ C.\ (u_1; u_2); u_3) \mid u_1, u_2, u_3 \in \mathcal{P}\}
\end{aligned}$$

  Sei $R = R_1 \cup R_2 \cup R_3 \cup R_4$. Wir zeigen, daß R die Anforderungen aus Definition 3.7 erfüllt. Zunächst betrachten wir R_1:

  - Wir nehmen $u_1; (u_2; u_3) \xrightarrow{\omega} u_{new}$ an und unterscheiden die folgenden Fälle:
    - $u_1 \xrightarrow{\alpha} u_1'$, $\omega = \alpha$ und $u_{new} = u_1'; (u_2; u_3)$, abgeleitet mit R_{12}. Dann können wir mit R_{12} folgern, daß $(u_1; u_2); u_3 \xrightarrow{\alpha} (u_1'; u_2); u_3$ gilt. Weiterhin gilt $(u_1'; (u_2; u_3), (u_1'; u_2); u_3) \in R$.
    - $u_1 \xrightarrow{\checkmark} u_1'$, $u_2 \xrightarrow{\alpha} u_2'$, $\omega = \alpha$ und $u_{new} = u_1'; (u_2'; u_3)$, abgeleitet mit R_{12}, R_{13}. Dann können wir mit R_{13}, R_{12} folgern, daß $(u_1; u_2); u_3 \xrightarrow{\alpha} (u_1'; u_2'); u_3$ gilt. Weiterhin gilt $(u_1'; (u_2'; u_3), (u_1'; u_2'); u_3) \in R$.
    - $u_1 \xrightarrow{\checkmark} u_1'$, $u_1' \xrightarrow{\alpha} u_1''$, $u_2 \xrightarrow{\alpha'} u_2'$, $\{\alpha, \alpha'\} = \{c?v, (c!v, C)\}$, $\omega = \tau$ und $u_{new} = \mathbf{ch}\ C.\ u_1''; (u_2'; u_3)$, abgeleitet mit R_{12}, R_{14}. Dann können wir mit R_{14}, R_{12} folgern, daß $(u_1; u_2); u_3 \xrightarrow{\tau} (\mathbf{ch}\ C.u_1''; u_2); u_3$ gilt. Mit C lokal in u_1 oder u_2 und $C \cap fc(u_1; (u_2; u_3)) = C \cap fc((u_1; u_2); u_3) = \emptyset$ wissen wir, daß $fc(u_3) \cap C = \emptyset$ gilt. Daher gilt $(\mathbf{ch}\ C.\ u_1''; (u_2'; u_3), (\mathbf{ch}\ C.\ u_1''; u_2); u_3) \in R$.
    - $u_1 \xrightarrow{\checkmark} u_1'$, $u_2 \xrightarrow{\checkmark} u_2'$, $u_3 \xrightarrow{\omega} u_3'$ und $u_{new} = u_1'; (u_2'; u_3')$, abgeleitet mit R_{13}. Dann können wir mit R_{13} folgern, daß $(u_1; u_2); u_3 \xrightarrow{\omega} (u_1'; u_2'); u_3'$ gilt. Weiterhin gilt $(u_1'; (u_2'; u_3'), (u_1'; u_2'); u_3') \in R$.
    - $u_1 \xrightarrow{\checkmark} u_1'$, $u_1' \xrightarrow{\alpha} u_1''$, $u_2 \xrightarrow{\checkmark} u_2'$, $u_3 \xrightarrow{\alpha'} u_3$, $\{\alpha, \alpha'\} = \{c?v, (c!v, C)\}$, $\omega = \tau$ und $u_{new} = \mathbf{ch}\ C.\ u_1''; (u_2'; u_3')$, abgeleitet mit R_{12}, R_{13}, R_{14}. Dann können wir mit R_{12} , R_{13} and R_{14} schließen, daß $(u_1; u_2); u_3 \xrightarrow{\tau} \mathbf{ch}\ C.\ (u_1''; u_2'); u_3$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\ u_1''; (u_2'; u_3'), \mathbf{ch}\ C.\ (u_1''; u_2'); u_3) \in R$.

  * $u_1 \xrightarrow{\checkmark} u_1', u_2 \xrightarrow{\checkmark} u_2', u_2 \xrightarrow{\alpha} u_2'', u_3 \xrightarrow{\alpha'} u_3', \{\alpha, \alpha'\} = \{c?v, (c!v, C)\}$, $\omega = \tau$ und $u_{new} = u_1'; (\mathbf{ch}\ C.\ u_2''; u_3')$, abgeleitet mit R_{12}, R_{13}, R_{14}. Dann können wir mit R_{13}, R_{12}, R_{14} schließen, daß $(u_1; u_2); u_3 \xrightarrow{\tau} \mathbf{ch}\ C.(u_1'; u_2''); u_3'$ gilt. Mit C lokal in u_2 oder u_3 und $fc(u_1; (u_2; u_3)) \cap C = fc((u_1; u_2); u_3) \cap C = \emptyset$ wissen wir, daß $fc(u_1) \cap C = \emptyset$ gilt. Daher gilt $(u_1'; (\mathbf{ch}\ C.\ u_2''; u_3'), \mathbf{ch}\ C.\ (u_1'; u_2''); u_3') \in R$.

- Der Beweis für die Annahme $(u_1; u_2); u_3 \xrightarrow{\omega} u_{new}$ erfolgt entsprechend.

Das Theorem kann für die Teilrelation R_4 analog bewiesen werden, indem die Techniken aus dem Beweis für die Kongruenz von $\sim_\beta$ bzgl. Kanalerzeugung in Theorem 3.8 eingesetzt werden. Für die Teilrelation R_2 wissen wir, daß $fc(u_3) \cap C = \emptyset$ gilt; daher beeinflußt die Restriktion von C nicht das Verhalten von u_3. Daher können wir auch hier die Techniken aus dem Kongruenzbeweis für $\sim_\beta$ bzgl. Kanalerzeugung anwenden. Dasselbe gilt für u_1 in der Teilrelation R_3.

- Axiom (3.7): $\{\vec{x} \leftarrow \vec{v}\}; t = t\langle \vec{v}/\vec{x} \rangle$.

  Sei $R = \{(\{\vec{x} \leftarrow \vec{v}\}; u, u\langle \vec{v}/\vec{x} \rangle) \mid u \in \mathcal{P}\} \cup \{(u, u) \mid u \in \mathcal{P}\}$. Wir unterscheiden die folgenden Fälle:

  - Wir nehmen $\{\vec{x} \leftarrow \vec{v}\}; u \xrightarrow{\omega} u'$ an. Dann wissen wir mit R_6, daß $u\langle \vec{v}/\vec{x} \rangle \xrightarrow{\omega} u'$ gilt. Weiterhin gilt $(u', u') \in R$.
  - Wir nehmen $u\langle \vec{v}/\vec{x} \rangle \xrightarrow{\omega} u'$ an. Dann wissen wir mit R_6, daß $\{\vec{x} \leftarrow \vec{v}\}; u \xrightarrow{\omega} u'$ gilt. Weiterhin gilt $(u', u') \in R$.

- Axiom (3.8): $t + \mathbf{0} = t$.

  Sei $R = \{(u + \mathbf{0}, u) \mid u \in \mathcal{P}\} \cup \{(u, u) \mid u \in \mathcal{P}\}$. Wir zeigen, daß R die in Definition 3.7 enthaltenen Anforderungen erfüllt.

  - Wir nehmen $u + \mathbf{0} \xrightarrow{\omega} u'$ an. Mit $\mathbf{0} \not\xrightarrow{}$ und R_9 können wir $u \xrightarrow{\omega} u'$ folgern. Weiterhin gilt $(u', u') \in R$.
  - Wir nehmen $u \xrightarrow{\omega} u'$ an. Mit R_9 können wir $u + \mathbf{0} \xrightarrow{\omega} u'$ folgern. Weiterhin gilt $(u', u') \in R$.

- Axiom (3.9): $t + u = u + t$.

  Sei $R = \{(t_0 + u_0, u_0 + t_0) \mid t_0, u_0 \in \mathcal{P}\} \cup \{(t_0, t_0) \mid t_0 \in \mathcal{P}\}$. Wir zeigen, daß R die in Definition 3.7 enthaltenen Anforderungen erfüllt.

  - Wir nehmen $t_0 + u_0 \xrightarrow{\omega} t_{new}$ an und unterscheiden die folgenden Fälle:
    * $t_0 \xrightarrow{\omega} t_0'$ und $t_{new} = t_0'$, abgeleitet mit R_9. Dann wissen wir mit R_{10}, daß $u_0 + t_0 \xrightarrow{\omega} t_0'$ gilt. Weiterhin gilt $(t_0', t_0') \in R$.
    * $u_0 \xrightarrow{\omega} u_0'$ und $t_{new} = u_0'$, abgeleitet mit R_{10}. Dann wissen wir mit R_9, daß $u_0 + t_0 \xrightarrow{\omega} u_0'$ gilt. Weiterhin gilt $(u_0', u_0') \in R$.

– Für die Annahme $u_0 + t_0 \xrightarrow{\omega} t_{new}$ erfolgt der Beweis analog.

- Axiom (3.10): $t_1 + (t_2 + t_3) = (t_1 + t_2) + t_3$.

  Sei
  $$\begin{aligned} R &= \{(u_1 + (u_2 + u_3), (u_1 + u_2) + u_3) \mid u_1, u_2, u_3 \in \mathcal{P}\} \\ &\quad \cup \{(u, u) \mid u \in \mathcal{P}\}. \end{aligned}$$

  Wir zeigen, daß R die Bisimulationsanforderungen aus Definition 3.7 erfüllt.

  – Wir nehmen $u_1 + (u_2 + u_3) \xrightarrow{\omega} u_{new}$ an und unterscheiden die folgenden Fälle:

    * $u_1 \xrightarrow{\omega} u_1'$ und $u_{new} = u_1'$, abgeleitet mit R_9. Dann können wir mit R_9 folgern, daß $(u_1 + u_2) + u_3 \xrightarrow{\omega} u_1'$ gilt. Weiterhin gilt $(u_1', u_1') \in R$.
    * $u_2 \xrightarrow{\omega} u_2'$ und $u_{new} = u_2'$, abgeleitet mit R_9, R_{10}. Dann können wir mit R_{10}, R_9 folgern, daß $(u_1 + u_2) + u_3 \xrightarrow{\omega} u_2'$ gilt. Weiterhin gilt $(u_2', u_2') \in R$.
    * $u_3 \xrightarrow{\omega} u_3'$ und $u_{new} = u_3'$, abgeleitet mit R_{10}. Dann können wir mit R_{10} folgern, daß $(u_1 + u_2) + u_3 \xrightarrow{\omega} u_3'$ gilt. Weiterhin gilt $(u_3', u_3') \in R$.

  – Der Beweis für die Annahme $(u_1 + u_2) + u_3 \xrightarrow{\omega} u_{new}$ erfolgt analog.

- Axiom (3.11): $t + t = t$.

  Sei $R = \{(u + u, u) \mid u \in \mathcal{P}\} \cup \{(u, u) \mid u \in \mathcal{P}\}$. Wir zeigen, daß R die Bisimulationsanforderungen aus Definition 3.7 erfüllt.

  – Wir nehmen $u + u \xrightarrow{\omega} u'$ an. Dann wissen wir R_9 bzw. R_{10}, daß $u \xrightarrow{\omega} u'$ gilt. Weiterhin gilt $(u', u') \in R$.

  – Wir nehmen $u \xrightarrow{\omega} u'$ an. Dann wissen wir mit R_9 bzw. R_{10}, daß $u + u \xrightarrow{\omega} u'$. Weiterhin gilt $(u', u') \in R$.

- Axiom (3.12): $(t_1 + t_2); t_3 = t_1; t_3 + t_2; t_3$.

  Sei
  $$\begin{aligned} R &= \{((u_1 + u_2); u_3, u_1; u_3 + u_2; u_3) \mid u_1, u_2, u_3 \in \mathcal{P}\} \\ &\cup \{(u, u) \mid u \in \mathcal{P}\}. \end{aligned}$$

  Wir zeigen, daß R die Anforderungen aus Definition 3.7 erfüllt.

  – Wir nehmen $(u_1 + u_2); u_3 \xrightarrow{\omega} u_{new}$ an und unterscheiden die folgenden Fälle:

    * $u_1 \xrightarrow{\alpha} u_1'$, $\omega = \alpha$ und $u_{new} = u_1'; u_3$, abgeleitet mit R_9, R_{12}. Dann können wir mit R_{12}, R_9 schließen, daß $u_1; u_3 + u_2; u_3 \xrightarrow{\alpha} u_1'; u_3$ gilt. Weiterhin gilt $(u_1'; u_3, u_1'; u_3) \in R$.

* $u_1 \xrightarrow{\checkmark} u_1'$, $u_3 \xrightarrow{\omega} u_3'$ und $u_{new} = u_1'; u_3'$, abgeleitet mit R_9, R_{13}. Dann können wir mit R_{13}, R_9 schließen, daß $u_1; u_3 + u_2; u_3 \xrightarrow{\omega} u_1'; u_3'$ gilt. Weiterhin gilt $(u_1'; u_3', u_1'; u_3') \in R$.
* $u_1 \xrightarrow{\checkmark} u_1'$, $u_1' \xrightarrow{\alpha} u_1''$, $u_3 \xrightarrow{\alpha'} u_3'$, $\{\alpha, \alpha'\} = \{c?v, (c!v, C)\}$, $\omega = \tau$ und $u_{new} = \mathbf{ch}\ C.\ u_1''; u_3'$, abgeleitet mit R_9, R_{14}. Dann können wir mit R_{14}, R_9 schließen, daß $u_1; u_3 + u_2; u_3 \xrightarrow{\tau} \mathbf{ch}\ C.\ u_1''; u_3'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\ u_1''; u_3', \mathbf{ch}\ C.\ u_1''; u_3') \in R$.
* $u_2 \xrightarrow{\alpha} u_2'$, $\omega = \alpha$ und $u_{new} = u_2'; u_3$, abgeleitet mit R_{10}, R_{12}. Dann können wir mit R_{12}, R_{10} schließen, daß $u_1; u_3 + u_2; u_3 \xrightarrow{\alpha} u_2'; u_3$ gilt. Weiterhin gilt $(u_2'; u_3, u_2'; u_3) \in R$.
* $u_2 \xrightarrow{\checkmark} u_2'$, $u_3 \xrightarrow{\omega} u_3'$ und $u_{new} = u_2'; u_3'$, abgeleitet mit R_{10}, R_{13}. Dann können wir mit R_{13}, R_{10} schließen, daß $u_1; u_3 + u_2; u_3 \xrightarrow{\omega} u_2'; u_3'$ gilt. Weiterhin gilt $(u_2'; u_3', u_2'; u_3') \in R$.
* $u_2 \xrightarrow{\checkmark} u_2'$, $u_2' \xrightarrow{\alpha} u_2''$, $u_3 \xrightarrow{\alpha'} u_3'$, $\{\alpha, \alpha'\} = \{c?v, (c!v, C)\}$, $\omega = \tau$ und $u_{new} = \mathbf{ch}\ C.\ u_2''; u_3'$, abgeleitet mit R_{10}, R_{14}. Dann können wir mit R_{14}, R_{10} schließen, daß $u_1; u_3 + u_2; u_3 \xrightarrow{\tau} \mathbf{ch}\ C.\ u_2''; u_3'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\ u_2''; u_3', \mathbf{ch}\ C.\ u_2''; u_3') \in R$.

– Der Beweis für die Annahme $u_1; u_3 + u_2; u_3$ erfolgt analog.

- Axiom (3.13): $\mathbf{ch}\ C.\ \mathbf{0} = \mathbf{0}$.

  Sei $R = \{(\mathbf{ch}\ C.\ \mathbf{0}, \mathbf{0})\}$. Mit $\mathbf{0} \not\xrightarrow{}$ und R_{15}, R_{16} wissen wir, daß $\mathbf{ch}\ C.\ \mathbf{0} \not\xrightarrow{}$ gilt. Daher erfüllt R die Bisimulationsanforderungen aus Definition 3.7.

- Axiom (3.14): $\mathbf{ch}\ C.\ \mathbf{1} = \mathbf{1}$.

  Sei $R = \{(\mathbf{ch}\ C.\ \mathbf{1}, \mathbf{1})\}$. Mit R_1 wissen wir, daß $\mathbf{1} \xrightarrow{\checkmark} \mathbf{1}$ gilt. Dann können wir mit R_{15} folgern, daß $\mathbf{ch}\ C.\mathbf{1} \xrightarrow{\checkmark} \mathbf{ch}\ C.\mathbf{1}$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\mathbf{1}, \mathbf{1}) \in R$; somit sind die Bisimulationsanforderungen aus Definition 3.7 erfüllt.

- Axiom (3.15): $\mathbf{ch}\ C.\ spawn(t) = spawn(\mathbf{ch}\ C.\ t)$.

  Sei $R = \{(\mathbf{ch}\ D.\ spawn(u), spawn(\mathbf{ch}\ D.\ u)) \mid u \in \mathcal{P}\}$. Wir zeigen, daß R die Anforderungen aus Definition 3.7 erfüllt.

  – Wir nehmen $\mathbf{ch}\ C.\ spawn(u) \xrightarrow{\omega} u_{new}$ an und unterscheiden die folgenden Fälle:

  * $u \xrightarrow{\alpha} u'$, $fc(\alpha) \cap C = \emptyset$, $\omega = \alpha$ und $u_{new} = \mathbf{ch}\ C.\ spawn(u')$, abgeleitet mit R_8, R_{15}. Weiterhin gilt $(\mathbf{ch}\ C.spawn(u'), spawn(\mathbf{ch}\ C.u'))$ $\in R$.
  * $u \xrightarrow{c!v,A} u'$, $c \notin C$, $A \cap C = \emptyset$, $fc(v) \cap C \neq \emptyset$, $\omega = A \cup (C \cap fc(\omega))$ und $u_{new} = \mathbf{ch}\ C \setminus fc(v).\ spawn(u')$, abgeleitet mit R_8, R_{16}. Dann können wir mit R_{16}, R_8 folgern, daß $spawn(\mathbf{ch}\ C.u) \xrightarrow{c!v, A \cup (C \cap fc(v))}$

$spawn(\mathbf{ch}\ C \setminus fc(v).\ u')$ gilt. Weiterhin gilt
$(\mathbf{ch}\ C \setminus fc(v).\ spawn(u'), spawn(\mathbf{ch}\ C \setminus fc(v).\ u')) \in R$.

* $spawn(u) \xrightarrow{\checkmark} spawn(u)$, $\omega = \checkmark$ und $u_{new} = \mathbf{ch}\ C.\ spawn(u)$, abgeleitet mit R_7, R_{15}. Dann können wir mit R_7 folgern, daß $spawn(\mathbf{ch}\ C.\ u) \xrightarrow{\checkmark} spawn(\mathbf{ch}\ C.\ u)$ gilt. Weiterhin gilt
$(\mathbf{ch}\ C.\ spawn(u), spawn(\mathbf{ch}\ C.\ u)) \in R$.

– Der Beweis für die Annahme $spawn(\mathbf{ch}\ C.\ u) \xrightarrow{\omega} u_{new}$ erfolgt analog.

- Axiom (3.16): $\mathbf{ch}\ C.\ t = t$ if $C \cap fc(t) = \emptyset$.

  Sei $R = \{(\mathbf{ch}\ C.\ u, u) \mid u \in \mathcal{P}, C \cap fc(u) = \emptyset\}$. Wir nehmen an, daß $u \xrightarrow{\omega} u'$ und $C \cap fc(u) = \emptyset$ gilt. Dann können wir mit R_9 folgern, daß $\mathbf{ch}\ C.\ u \xrightarrow{\omega} \mathbf{ch}\ C.\ u'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\ u', u') \in R$ und somit erfüllt R die Anforderungen aus Definition 3.7.

- Axiom (3.17): $\mathbf{ch}\ C.\ \mathbf{ch}\ C'.\ t = \mathbf{ch}\ C'.\ \mathbf{ch}\ C.\ t$.

  Sei $R = \{(\mathbf{ch}\ D.\ \mathbf{ch}\ D'.\ u, \mathbf{ch}\ D'.\ \mathbf{ch}\ D.\ u) \mid u \in \mathcal{P}\}$. Wir zeigen, daß R die Bisimulationsanforderungen aus Definition 3.7 erfüllt. Für den Beweis nehmen wir an, daß $C \cap C' = \emptyset$ gilt; andernfalls wenden wir α-Konversion an.

  – Wir nehmem $\mathbf{ch}\ C.\ \mathbf{ch}\ C'.\ u \xrightarrow{\omega} u_{new}$ an und unterscheiden die folgenden Fälle:

    * $u \xrightarrow{\omega} u'$, $fc(\omega) \cap (C \cup C') = \emptyset$ und $u_{new} = \mathbf{ch}\ C.\mathbf{ch}\ C'.u'$, abgeleitet mit R_{15}. Dann können wir mit R_{15} schließen, daß $\mathbf{ch}\ C'.\mathbf{ch}\ C.u \xrightarrow{\omega} \mathbf{ch}\ C'.\mathbf{ch}\ C.u'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\mathbf{ch}\ C'.u', \mathbf{ch}\ C'.\mathbf{ch}\ C.u') \in R$.
    * $u \xrightarrow{c!v,A} u'$, $c \notin (C \cup C')$, $A \cap (C \cup C') = \emptyset$, $fc(v) \cap (C \cup C') \neq \emptyset$, $\omega = (c!v, A \cup ((C \cup C') \cap fc(v)))$ und $u_{new} = \mathbf{ch}\ C \setminus fc(v).\ \mathbf{ch}\ C' \setminus fc(v).\ u'$, abgeleitet mit R_{16}. Dann können wir mit R_{16} schließen, daß $\mathbf{ch}\ C'.\ \mathbf{ch}\ C.\ u \xrightarrow{c!v, A \cup ((C \cup C') \cap fc(v))} \mathbf{ch}\ C' \setminus fc(v).\ \mathbf{ch}\ C \setminus fc(v).\ u'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C \setminus fc(v).\ \mathbf{ch}\ C' \setminus fc(v).\ u', \mathbf{ch}\ C' \setminus fc(v).\mathbf{ch}\ C \setminus fc(v).u') \in R$ (wir weisen darauf hin, daß $fc(v) \cap C \neq \emptyset \neq fc(v) \cap C'$ möglich ist).

  – Der Beweis für die Annahme $\mathbf{ch}\ C'.\ \mathbf{ch}\ C.\ u \xrightarrow{\omega} u_{new}$ erfolgt analog.

- Axiom (3.18): $\mathbf{ch}\ C.\ t + u = \mathbf{ch}\ C.\ t + \mathbf{ch}\ C.\ u$.

  Sei

$$\begin{aligned} R \;=\; & \{(\mathbf{ch}\ D.\ t_0 + u_0, \mathbf{ch}\ D.\ t_0 + \mathbf{ch}\ D.\ u_0) \mid t_0, u_0 \in \mathcal{P}\} \\ & \cup \{(t_0, t_0) \mid t_0 \in \mathcal{P}\}. \end{aligned}$$

  Wir zeigen, daß sie die Bisimulationsanforderungen aus Definition 3.7 erfüllt.

– Wir nehmen $\mathbf{ch}\ C.\, t_0 + u_0 \xrightarrow{\omega} t_{new}$ an und unterscheiden die folgenden Fälle:

* $t_0 \xrightarrow{\omega} t_0'$, $fc(\omega) \cap C = \emptyset$ und $t_{new} = \mathbf{ch}\ C.t_0'$, abgeleitet mit R_9, R_{15}. Dann können wir mit R_{15}, R_9 schließen, daß $\mathbf{ch}\ C.t_0 + \mathbf{ch}\ C.u_0 \xrightarrow{\omega} \mathbf{ch}\ C.\, t_0'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\, t_0', \mathbf{ch}\ C.\, t_0') \in R$.
* $u_0 \xrightarrow{\omega} u_0'$, $fc(\omega) \cap C = \emptyset$ und $t_{new} = \mathbf{ch}\ C.\, u_0'$, abgeleitet mit R_{10}, R_{15}. Dann können wir mit R_{15}, R_{10} schließen, daß $\mathbf{ch}\ C.\, t_0 + \mathbf{ch}\ C.\, u_0 \xrightarrow{\omega} \mathbf{ch}\ C.\, u_0'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\, u_0', \mathbf{ch}\ C.\, u_0') \in R$.
* $t_0 \xrightarrow{c!v,A} t_0'$, $A \cap C = \emptyset$, $c \notin C$, $fc(v) \cap C \neq \emptyset$, $\omega = c!v$, $A \cup (C \cap fc(v))$ und $t_{new} = \mathbf{ch}\ C \setminus fc(v).\, t_0'$, abgeleitet mit R_9, R_{16}. Dann können wir mit R_{16}, R_9 schließen, daß $\mathbf{ch}\ C.\, t_0 + \mathbf{ch}\ C.\, u_0 \xrightarrow{c!v,A \cup (C \cap fc(v))} \mathbf{ch}\ C \setminus fc(v).t_0'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C \setminus fc(v).t_0', \mathbf{ch}\ C \setminus fc(v).t_0') \in R$.
* $u_0 \xrightarrow{c!v,A} u_0'$, $A \cap C = \emptyset$, $c \notin C$, $fc(v) \cap C \neq \emptyset$, $\omega = c!v, A \cup (C \cap fc(v))$ und $t_{new} = \mathbf{ch}\ C \setminus fc(v).\, u_0'$, abgeleitet mit R_{10}, R_{16}. Dann können wir mit R_{16}, R_{10} schließen, daß $\mathbf{ch}\ C.\, t_0 + \mathbf{ch}\ C.u_0 \xrightarrow{c!v,A \cup (C \cap fc(v))} \mathbf{ch}\ C \setminus fc(v).u_0'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C \setminus fc(v).\, u_0', \mathbf{ch}\ C \setminus fc(v).\, u_0') \in R$.

– Der Beweis für die Annahme $\mathbf{ch}\ C.t_0 + \mathbf{ch}\ C.u_0 \xrightarrow{\omega} t_{new}$ erfolgt analog.

- Axiom (3.19): $\mathbf{ch}\ C.\, c?x; t = \mathbf{0}$ falls $c \in C$.

  Sei $R = \{(\mathbf{ch}\ C.\, c?x; u, \mathbf{0}) \mid u \in \mathcal{P}\}$. Wir wissen, daß $\mathbf{0} \not\longrightarrow$ gilt. Weiterhin können wir mit $c \in C$ und R_5, R_{15}, R_{16} schließen, daß $\mathbf{ch}\ C.\, c?x; u \not\longrightarrow$ gilt. Daher erfüllt R die Bisimulationsanforderungen aus Definition 3.7.

- Axiom (3.20): $\mathbf{ch}\ C.\, c!v; t = \mathbf{0}$ if $c \in C$.

  Sei $R = \{(\mathbf{ch}\ C.\, c!v; u, \mathbf{0}) \mid u \in \mathcal{P}, c \in C\}$. Wir wissen, daß $\mathbf{0} \not\longrightarrow$ gilt. Weiterhin wissen wir mit R_4, R_{15}, R_{16}, daß $\mathbf{ch}\ C.\, c!v; u \not\longrightarrow$ gilt. Daher erfüllt R die Bisimulationsanforderungen aus Definition 3.7.

- Axiom (3.21): $\mathbf{ch}\ C.\, \gamma; t = \gamma; \mathbf{ch}\ C.\, t$ if $C \cap fc(\gamma) = \emptyset$.

  Sei $R = \{(\mathbf{ch}\ C.\, \gamma; u, \gamma; \mathbf{ch}\ C.\, u) \mid u \in \mathcal{P}, C \cap fc(\gamma) = \emptyset\} \cup$
  $\{(\mathbf{ch}\ C.\, \mathbf{1}; u, \mathbf{1}; \mathbf{ch}\ C.\, u) \mid u \in \mathcal{P}\}$. Wir zeigen, daß R die Bisimulationsanforderungen aus Definition 3.7 erfüllt. Mit R_{12}, R_{15} und $C \cap fc(\gamma) = \emptyset$ können wir folgern, daß $\mathbf{ch}\ C.\gamma; u \xrightarrow{\gamma} \mathbf{ch}\ C.\mathbf{1}; u$ gilt. Ebenso können wir mit R_{12} folgern, daß $\gamma; \mathbf{ch}\ C.\, u \xrightarrow{\gamma} \mathbf{1}; \mathbf{ch}\ C.\, u$ gilt. Für $\gamma = [E]$ und $[\![E]\!] = true$ können wir mit $fc(\gamma) \cap C = \emptyset$, R_3, R_{12}, R_{15} schließen, daß $\mathbf{ch}\ C.\, \gamma; u \xrightarrow{\tau} \mathbf{ch}\ C.\, \mathbf{1}; u$ und $\gamma; \mathbf{ch}\ C.\, u \xrightarrow{\tau} \mathbf{1}; \mathbf{ch}\ C.\, u$ gilt. Weiterhin gilt
  $(\mathbf{ch}\ C.\, \mathbf{1}; u, \mathbf{1}; \mathbf{ch}\ C.\, u) \in R$. Der Beweis für die Teilrelation
  $\{(\mathbf{ch}\ C.\, \mathbf{1}; u, \mathbf{1}; \mathbf{ch}\ C.\, u) \mid u \in \mathcal{P}\}$ erfolgt analog zum Beweis für Axiom (3.3).

- Axiom (3.22): $\mathbf{ch}\ C.\, c?x; t = c?x; \mathbf{ch}\ C.\, t$ falls $c \notin C$.

  Sei $R = \{(\mathbf{ch}\ C.\, c?x; u, c?x; \mathbf{ch}\ C.\, u) \mid u \in \mathcal{P}, c \notin C\} \cup \sim_\beta$. Wir zeigen, daß R die Bisimulationsanforderungen aus Definition 3.7 erfüllt.

  - Wir nehmen $\mathbf{ch}\ C.\, c?x; u \xrightarrow{\omega} u_{new}$ an. Dann gilt $\omega = c?v$, $fc(v) \cap C = \emptyset$ und $u_{new} = \mathbf{ch}\ C.\, u\langle v/x\rangle$, abgeleitet mit R_5, R_{15}. Sei $v' =_\beta v$ beliebig. Dann können wir mit R_5 ableiten, daß $c?x; \mathbf{ch}\ C.\, u \xrightarrow{c?v'} (\mathbf{ch}\ C.\, u)\langle v'/x\rangle$ gilt. Aus $v' =_\beta v$ folgt $fc(v') = fc(v)$. Daher gilt $fc(v') \cap C = \emptyset$ und $(\mathbf{ch}\ C.\, u)\langle v'/x\rangle = \mathbf{ch}\ C.\, u\langle v'/x\rangle$. Mit Lemma 3.9 und Theorem 3.8 können wir $\mathbf{ch}\ C.\, u\langle v/x\rangle \sim_\beta \mathbf{ch}\ C.\, u\langle v'/x\rangle$ folgern. Daher gilt $(\mathbf{ch}\ C.\, u\langle v/x\rangle, \mathbf{ch}\ C.\, u\langle v'/x\rangle) \in R$.
  - Wir nehmen $c?x; \mathbf{ch}\ C.\, u \xrightarrow{\omega} u_{new}$ an. Dann gilt $\omega = c?v$ und $u_{new} = (\mathbf{ch}\ C.\, u)\langle v/x\rangle$, abgeleitet mit R_5. Wir nehmen an, daß $fc(v) \cap C = \emptyset$ gilt; andernfalls wenden wir α-Konversion an. Dann können wir schließen, daß $(\mathbf{ch}\ C.\, u)\langle v/x\rangle = \mathbf{ch}\ C.\, u\langle v/x\rangle$ gilt. Sei $v' =_\beta v$ beliebig. Dann wissen wir, daß $fc(v) = fc(v')$ gilt. Hieraus folgt $fc(v') \cap C = \emptyset$ und wir können mit R_5, R_{15} folgern, daß $\mathbf{ch}\ C.\, c?x; u \xrightarrow{c?v'} \mathbf{ch}\ C.\, u\langle v'/x\rangle$ gilt. Dann können wir mit Lemma 3.9 und Theorem 3.8 folgern, $\mathbf{ch}\ C.\, u\langle v/x\rangle \sim_\beta \mathbf{ch}\ C.\, u\langle v'/x\rangle$ gilt. Daher gilt $(\mathbf{ch}\ C.\, u\langle v/x\rangle, \mathbf{ch}\ C.\, u\langle v'/x\rangle) \in R$.

- Axiom (3.23): $(\mathbf{ch}\ C.\, t); u = \mathbf{ch}\ C.\, t; u$ falls $C \cap fc(u) = \emptyset$.

  Sei $R = \{(\mathbf{ch}\ E.\, (\mathbf{ch}\ D.\, t_0); u_0, \mathbf{ch}\ D \cup E.\, t_0; u_0)\}$. Wir zeigen, daß R die Bisimulationsanforderungen aus Definition 3.7 erfüllt. Wir führen den Beweis für $E = \emptyset$. Für den Fall $E \neq \emptyset$ können die Techniken aus dem Beweis für die Kongruenzeigenschaft von $\sim_\beta$ bzgl. Kanalerzeugung in Theorem 3.8 verwendet werden.

  - Wir nehmen $(\mathbf{ch}\ D.t_0); u_0 \xrightarrow{\omega} t_{new}$ an und unterscheiden die folgenden Fälle:
    - $t_0 \xrightarrow{\omega} t_0'$, $fc(\omega) \cap D = \emptyset$ und $t_{new} = (\mathbf{ch}\ D.\, t_0'); u_0$, abgeleitet mit R_{15}, R_{12}. Dann können wir mit R_{12}, R_{15} folgern, daß $\mathbf{ch}\ D.\, t_0; u_0 \xrightarrow{\omega} \mathbf{ch}\ D.\, t_0'; u_0$ gilt. Weiterhin gilt $((\mathbf{ch}\ D.\, t_0'); u_0, \mathbf{ch}\ D.\, t_0'; u_0) \in R$.
    - $t_0 \xrightarrow{c!v,A} t_0'$, $c \notin D$, $D \cap A = \emptyset$, $fc(v) \cap D \neq \emptyset$, $\omega = A \cup (D \cap fc(v))$ und $t_{new} = \mathbf{ch}\ (.\, D \setminus fc(v) t_0'); u_0$, abgeleitet mit R_{16}, R_{12}. Dann können wir mit R_{12}, R_{16} folgern, daß $\mathbf{ch}\ D.\, t_0; u_0 \xrightarrow{c!v,A\cup(D\cap fc(v))} \mathbf{ch}\ D \setminus fc(v).t_0'; u_0$ gilt. Weiterhin gilt $((\mathbf{ch}\ D \setminus fc(v).t_0'); u_0, \mathbf{ch}\ D \setminus fc(v).\, t_0'; u_0) \in R$.
    - $t_0 \xrightarrow{\checkmark} t_0'$, $u_0 \xrightarrow{\omega} u_0'$ und $t_{new} = (\mathbf{ch}\ D.\, t_0); u_0'$, abgeleitet mit R_{15}, R_{13}. Dann können wir mit R_{13}, R_{15} und $D \cap fc(u_0) = \emptyset$ folgern,

daß $\mathbf{ch}\ D.\, t_0; u_0 \xrightarrow{\omega} \mathbf{ch}\ D.\, t_0'; u_0'$ gilt. Weiterhin gilt $((\mathbf{ch}\ D.\, t_0); u_0', \mathbf{ch}\ D.\, t_0'; u_0') \in R$.

* $t_0 \xrightarrow{\checkmark} t_0', t_0' \xrightarrow{\alpha} t_0'', u_0 \xrightarrow{\alpha}{}' u_0', fc(\alpha) \cap D = \emptyset, \{\alpha, \alpha'\} = \{c?v, (c!v, A)\}$, $\omega = \tau$ und $t_{new} = \mathbf{ch}\ A.\, (\mathbf{ch}\ D.\, t_0''); u_0'$, abgeleitet mit R_{15}, R_{14}. Dann können wir mit R_{14}, R_{15} folgern, daß $\mathbf{ch}\ D. t_0, u_0 \xrightarrow{\tau} \mathbf{ch}\ D \cup A. t_0''; u_0'$ gilt. Weiterhin gilt $(\mathbf{ch}\ A. (\mathbf{ch}\ D. t_0''); u_0', \mathbf{ch}\ D \cup A. t_0''; u_0') \in R$.
* $t_0 \xrightarrow{\checkmark} t_0', t_0' \xrightarrow{c!v,A} t_0'', u_0 \xrightarrow{c?v} u_0', c \notin D, A \cap D = \emptyset, fc(v) \cap D \neq \emptyset$, $\omega = \tau$ und $t_{new} = \mathbf{ch}\ A \cup (D \cap fc(v)).\, (\mathbf{ch}\ D \setminus fc(v).\, t_0''; u_0')$, abgeleitet mit R_{16}, R_{14}. Dann können wir mit R_{14}, R_{16} und $(D \setminus fc(v)) \cup (A \cup (D \cap fc(v))) = (A \cup D)$ folgern, daß $\mathbf{ch}\ D.\, t_0; u_0 \xrightarrow{\tau} \mathbf{ch}\ A \cup D.\, t_0''; u_0'$ gilt. Weiterhin gilt $(\mathbf{ch}\ A \cup (D \cap fc(v)).\, (\mathbf{ch}\ D \setminus fc(v).\, t_0''); u_0', \mathbf{ch}\ A \cup D.\, t_0''; u_0') \in R$.

– Wir nehmen $\mathbf{ch}\ D.\, t_0; u_0 \xrightarrow{\omega} t_{new}$ an und unterscheiden die folgenden Fälle:

* $t_0 \xrightarrow{\omega} t_0', fc(\omega) \cap D = \emptyset$ und $t_{new} = \mathbf{ch}\ D. t_0'; u_0$, abgeleitet mit R_{12}, R_{15}. Dann können wir mit R_{15}, R_{12} folgern, daß $(\mathbf{ch}\ D.\, t_0); u_0 \xrightarrow{\omega} (\mathbf{ch}\ D.\, t_0'); u_0$ gilt. Weiterhin gilt $((\mathbf{ch}\ D.\, t_0'); u_0, \mathbf{ch}\ D.\, t_0'; u_0) \in R$.
* $t_0 \xrightarrow{c!v,A} t_0'$, $c \notin D$, $A \cap D = \emptyset$, $fc(v) \cap D \neq \emptyset$, $\omega = c!v, A \cup (D \cap fc(v))$ und $t_{new} = \mathbf{ch}\ D \setminus fc(v). t_0'; u_0$, abgeleitet mit R_{12}, R_{16}. Dann können wir mit R_{16}, R_{12} folgern, daß $(\mathbf{ch}\ D.\, t_0); u_0 \xrightarrow{c!v, A \cup (D \cap fc(v))} (\mathbf{ch}\ D \setminus fc(v). t_0'); u_0$ gilt. Weiterhin gilt $((\mathbf{ch}\ D \setminus fc(v). t_0'); u_0, \mathbf{ch}\ D \setminus fc(v).\, t_0'; u_0) \in R$.
* $t_0 \xrightarrow{\checkmark} t_0'$, $u_0 \xrightarrow{\omega} u_0'$ und $t_{new} = \mathbf{ch}\ D.\, t_0'; u_0'$, abgeleitet mit R_{13}, R_{15} und $D \cap fc(u) = \emptyset$. Dann können wir mit R_{15}, R_{13} folgern, daß $(\mathbf{ch}\ D.\, t_0); u_0 \xrightarrow{\omega} (\mathbf{ch}\ D.\, t_0'); u_0'$ gilt. Weiterhin gilt $((\mathbf{ch}\ D.\, t_0'); u_0', \mathbf{ch}\ D.\, t_0'; u_0') \in R$.
* $t_0 \xrightarrow{\checkmark} t_0', t_0' \xrightarrow{\alpha} t_0'', fc(\alpha) \cap D = \emptyset, \{\alpha, \alpha'\} = \{c?v, (c!v, A)\}, \omega = \tau$ und $t_{new} = \mathbf{ch}\ A \cup D.\, t_0''; u_0'$, abgeleitet mit R_{14}, R_{15}. Dann können wir mit R_{15}, R_{14} folgern, daß $(\mathbf{ch}\ D.\, t_0); u_0 \xrightarrow{\tau} \mathbf{ch}\ A.\, (\mathbf{ch}\ D.\, t_0''); u_0'$ gilt. Weiterhin gilt $(\mathbf{ch}\ A.\, (\mathbf{ch}\ D.\, t_0''); u_0', \mathbf{ch}\ A \cup D.\, t_0''; u_0') \in R$.
* $t_0 \xrightarrow{\checkmark} t_0', t_0' \xrightarrow{c!v,A} t_0'', c \notin D, A \cap D = \emptyset, fc(v) \cap D \neq \emptyset, \omega = \tau$ und $t_{new} = \mathbf{ch}\ D \cup A.\, t_0''; u_0'$, abgeleitet mit R_{14}, R_{16} und $(D \setminus fc(v)) \cup (A \cup (D \cap fc(v))) = D \cup A$. Dann können wir mit R_{16}, R_{14} folgern, daß $(\mathbf{ch}\ D.\, t_0); u_0 \xrightarrow{\tau} \mathbf{ch}\ A \cup (D \cap fc(v)).\, (\mathbf{ch}\ D \setminus fc(v).\, t_0''); u_0'$ gilt. Weiterhin gilt $(\mathbf{ch}\ A \cup (D \cap fc(v)).\, (\mathbf{ch}\ D \setminus fc(v).\, t_0''); u_0', \mathbf{ch}\ D \cup A.\, t_0''; u_0') \in R$.

- Axiom (3.24): $t; \mathbf{ch}\ C.\, u = \mathbf{ch}\ C.\, t; u$ falls $C \cap fc(t) = \emptyset$.

Sei $R = \{(\mathbf{ch}\ E.\, t_0; \mathbf{ch}\ D.\, u_0, \mathbf{ch}\ E \cup D.\, t_0; u_0)\}$. Der Beweis, daß R eine Bisimulation ist, erfolgt analog zum Beweis für Axiom (3.23).

- Axiom (3.25): $P(\vec{v}) = \tau; t\langle\vec{v}/\vec{x}\rangle$ falls $\Theta : P(\vec{x} : \vec{T}) \mapsto t$.

  Sei
  $$\begin{aligned} R &= \{(P(\vec{v}), \tau; u\langle\vec{v}/\vec{x}\rangle) \mid u \in \mathcal{P}, \Theta : P(\vec{x} : \vec{T}) \mapsto u\} \\ &\cup \{(u, \mathbf{1}; u) \mid u \in \mathcal{P}\}. \end{aligned}$$

  Mit R_{11} können wir $P(\vec{v}) \xrightarrow{\tau} u\langle\vec{v}/\vec{x}\rangle$ ableiten. Mit R_2, R_{12} können wir $\tau; u\langle\vec{v}/\vec{x}\rangle \xrightarrow{\tau} \mathbf{1}; u\langle\vec{v}/\vec{x}\rangle$ ableiten. Weiterhin gilt $(u\langle\vec{v}/\vec{x}\rangle, \mathbf{1}; u\langle\vec{v}/\vec{x}\rangle)$ $\in R$. Der Beweis für die Teilrelation $\{(u, \mathbf{1}; u) \mid u \in \mathcal{P}\}$ erfolgt analog zum Beweis für Axiom (3.3). Daher erfüllt R die Bisimulationsanforderungen aus Definition 3.7.

- Axiom (3.26): $spawn(\mathbf{0}) = \mathbf{1}$.

  Sei $R = \{(spawn(\mathbf{0}), \mathbf{1})\}$. Wir zeigen, daß R die Bisimulationsanforderungen aus Definition 3.7 erfüllt.

  - Wir nehmen $spawn(\mathbf{0}) \xrightarrow{\omega} t'$ an. Aus $\mathbf{0} \not\longrightarrow$ und mit R_7 folgt $\omega = \checkmark$ und $t' = spawn(\mathbf{0})$. Ebenso wissen wir mit R_1, daß $\mathbf{1} \xrightarrow{\checkmark} \mathbf{1}$ gilt. Weiterhin gilt $(spawn(\mathbf{0}), \mathbf{1}) \in R$.
  - Für die Annahme $\mathbf{1} \xrightarrow{\omega} t'$ analog.

- Axiom (3.27): $spawn(t); spawn(u) = spawn(u); spawn(t)$.
  Sei
  $$\begin{aligned} R &= \{(\mathbf{ch}\ D.\, spawn(t_0); spawn(u_0), \mathbf{ch}\ D.\, spawn(u_0); spawn(t_0)) \mid \\ &\quad t_0; u_0 \in \mathcal{P}\}. \end{aligned}$$

  Wir zeigen, daß R die Bisimulationseigenschaften von Definition 3.7 erfüllt. Wir beweisen den Fall $D = \emptyset$. Für $D \neq \emptyset$ können die Techniken aus dem Beweis der Kongruenzeigenschaft von $\sim_\beta$ bzgl. Kanalerzeugung aus Theorem 3.8 verwendet werden.

  - Wir nehmen $spawn(t_0); spawn(u_0) \xrightarrow{\omega} t_{new}$ an und unterscheiden die folgenden Fälle:
    - $t_0 \xrightarrow{\alpha} t_0'$, $\omega = \alpha$ und $t_{new} = spawn(t_0'); spawn(u_0')$, abgeleitet mit R_8, R_{12}. Dann können wir mit R_7, R_8, R_{13} folgern, daß $spawn(u_0); spawn(t_0) \xrightarrow{\alpha} spawn(u_0); spawn(t_0')$ gilt. Weiterhin gilt $(spawn(t_0'); spawn(u_0), spawn(u_0); spawn(t_0')) \in R$.
    - $u_0 \xrightarrow{\alpha} u_0'$, $\omega = \alpha$ und $t_{new} = spawn(t_0); spawn(u_0')$, abgeleitet mit R_7, R_8, R_{13}. Dann können wir mit R_8, R_{12} folgern, daß $spawn(u_0); spawn(t_0) \xrightarrow{\alpha} spawn(u_0'); spawn(t_0)$ gilt. Weiterhin gilt $(spawn(t_0); spawn(u_0'), spawn(u_0'); spawn(t_0)) \in R$.

  - $t_0 \xrightarrow{\alpha} t_0'$, $u_0 \xrightarrow{\alpha'} u_0'$, $\{\alpha, \alpha'\} = \{c?v, (c!v, C)\}$, $\omega = \tau$ und $t_{new} =$ **ch** $C.spawn(t_0'); spawn(u_0')$, abgeleitet mit R_7, R_8, R_{14}. Dann können wir mit R_7, R_8, R_{14} ableiten, daß $spawn(u_0); spawn(t_0) \xrightarrow{\tau}$ **ch** $C.spawn(u_0'); spawn(t_0')$ gilt. Weitrhin gilt (**ch** $C.spawn(t_0'); spawn(u_0')$, **ch** $C.spawn(u_0'); spawn(t_0')) \in R$.
  - $\omega = \checkmark$ and $t_{new} = spawn(t_0); spawn(u_0)$, abgeleitet mit R_7, R_{13}. Dann können wir mit R_7, R_{13} ableiten, daß $spawn(u_0); spawn(t_0) \xrightarrow{\checkmark} spawn(u_0); spawn(t_0)$ gilt. Weiterhin gilt $(spawn(t_0); spawn(u_0), spawn(u_0); spawn(t_0)) \in R$.

- Für die Annahme $spawn(u_0); spawn(t_0) \xrightarrow{\omega} t_{new}$ analog.

- Axiom (3.28): $spawn(t); spawn(u) = spawn(spawn(t); u)$.
  Sei

$$R = \{(\textbf{ch } D.\ spawn(t_0); spawn(u_0), spawn(\textbf{ch } D.\ spawn(t_0); u_0) \mid t_0, u_0 \in \mathcal{P}\}.$$

  Wir zeigen, daß R die Bisimulationsanforderungen aus Definition 3.7 erfüllt. Wir führen den Beweis für den Fall $D = \emptyset$. Für $D \neq \emptyset$ erfolgt der Beweis analog zum Beweis für Axiom (3.15).

  - Wir nehmen $spawn(t_0); spawn(u_0) \xrightarrow{\omega} t_{new}$ an und unterscheiden die folgenden Fälle:
    - $t_0 \xrightarrow{\alpha} t_0'$, $\omega = \alpha$ und $t_{new} = spawn(t_0'); spawn(u_0)$, abgeleitet mit R_8, R_{12}. Dann können wir mit R_8, R_{12} ableiten, daß $spawn(spawn(t_0); u_0) \xrightarrow{\alpha} spawn(spawn(t_0'); u_0)$ gilt. Weiterhin gilt $(spawn(t_0'); spawn(u_0), spawn(spawn(t_0'); u_0)) \in R$.
    - $u_0 \xrightarrow{\alpha} u_0'$, $\omega = \alpha$ und $t_{new} = spawn(t_0); spawn(u_0')$, abgeleitet mit R_7, R_8, R_{13}. Dann können wir mit R_7, R_{13}, R_8 ableiten, daß eine Transition $spawn(spawn(t_0); u_0) \xrightarrow{\alpha} spawn(spawn(t_0); u_0')$ möglich ist. Weiterhin gilt $(spawn(t_0); spawn(u_0'), spawn(spawn(t_0); u_0')) \in R$.
    - $t_0 \xrightarrow{\alpha} t_0'$, $u_0 \xrightarrow{\alpha'} u_0'$, $\{\alpha, \alpha'\} = \{c?v, (c!v, C)\}$, $\omega = \tau$ und $t_{new} =$ **ch** $C.spawn(t_0'); spawn(u_0')$, abgeleitet mit R_7, R_8, R_{14}. Dann können wir mit R_7, R_8, R_{14} schließen, daß $spawn(spawn(t_0); u_0) \xrightarrow{\tau} spawn(\textbf{ch } C.\ spawn(t_0'); u_0')$ gilt. Weiterhin gilt $(\textbf{ch } C.\ spawn(t_0'); spawn(u_0'), spawn(\textbf{ch } C.\ spawn(t_0'); u_0')) \in R$.
    - $\omega = \checkmark$ und $t_{new} = spawn(t_0); spawn(u_0)$, abgeleitet mit R_7, R_{13}. Ebenso können wir mit R_7 folgern, daß $spawn(spawn(t_0); u_0) \xrightarrow{\checkmark} spawn(spawn(t_0); u_0)$ gilt. Weiterhin gilt $(spawn(t_0); spawn(u_0), spawn(spawn(t_0); u_0)) \in R$.
  - Für die Annahme $spawn(spawn(t_0); u_0) \xrightarrow{\omega} t_{new}$ analog.

- Axiom (3.29): $spawn(t_1; spawn(t_2) + t_3) = spawn(t_1; t_2 + t_3)$.
  Sei

$$\begin{aligned} R_1 &= \{(spawn(u_1; spawn(u_2) + u_3), spawn(u_1; u_2 + u_3)) \mid \\ &\quad u_1, u_2, u_3 \in \mathcal{P}\} \\ R_2 &= \{(spawn(\mathbf{ch}\ C.\ u_1; spawn(u_2)), spawn(\mathbf{ch}\ C.\ u_1; u_2)) \mid \\ &\quad u_1, u_2 \in \mathcal{P}\} \\ R_3 &= \{(u, u) \mid u \in \mathcal{P}\}\ . \end{aligned}$$

  Sei $R = R_1 \cup R_2 \cup R_3$. Wir zeigen, daß R die Bisimulationsanforderungen aus Definition 3.7 erfüllt. Zunächst betrachten wir die Teilrelation R_1.

  - Wir nehmen $spawn(u_1; spawn(u_2) + u_3) \xrightarrow{\omega} t_{new}$ an und unterscheiden die folgenden Fälle:
    * $u_1 \xrightarrow{\alpha} u_1'$, $\omega = \alpha$ und $t_{new} = spawn(u_1'; spawn(u_2))$, abgeleitet mit R_{12}, R_9, R_8. Dann können wir mit R_{12}, R_9, R_8 ableiten, daß $spawn(u_1; u_2 + u_3) \xrightarrow{\alpha} spawn(u_1'; u_2)$ gilt. Weiterhin gilt $(spawn(u_1'; spawn(u_2)), spawn(u_1'; u_2)) \in R$.
    * $u_1 \xrightarrow{\checkmark} u_1'$, $u_2 \xrightarrow{\alpha} u_2'$, $\omega = \alpha$ und $t_{new} = spawn(u_1'; spawn(u_2'))$, abgeleitet mit R_8, R_{13}, R_9. Dann können wir mit R_{13}, R_9, R_8 ableiten, daß $spawn(u_1; u_2 + u_3) \xrightarrow{\alpha} spawn(u_1'; u_2')$ gilt. Weiterhin gilt $(spawn(u_1'; spawn(u_2')), spawn(u_1'; u_2')) \in R$.
    * $u_1 \xrightarrow{\checkmark} u_1'$, $u_1' \xrightarrow{\alpha} u_1''$, $u_2 \xrightarrow{\alpha'} u_2'$, $\{\alpha, \alpha'\} = \{c?v, (c!v, C)\}$, $\omega = \tau$ und $t_{new} = spawn(\mathbf{ch}\ C.\ u_1''; spawn(u_2'))$, abgeleitet mit R_8, R_{14}, R_9. Dann können wir mit R_{14}, R_9, R_8 ableiten, daß $spawn(u_1; u_2 + u_3) \xrightarrow{\tau} spawn(\mathbf{ch}\ C.\ u_1''; u_2')$ gilt. Weiterhin gilt $(spawn(\mathbf{ch}\ C.\ u_1''; spawn(u_2')), spawn(\mathbf{ch}\ C.\ u_1''; u_2')) \in R$.
    * $u_3 \xrightarrow{\alpha} u_3'$, $\omega = \alpha$ und $t_{new} = spawn(u_3')$, abgeleitet mit R_{10}, R_8. Dann können wir mit R_{10}, R_8 ableiten, daß $spawn(u_1; u_2 + u_3) \xrightarrow{\alpha} spawn(u_3')$ gilt. Weiterhin gilt $(spawn(u_3), spawn(u_3)) \in R$.
    * $\omega = \checkmark$ und $t_{new} = spawn(u_1; spawn(u_2) + u_3)$, abgeleitet mit R_7. Ebenso können wir mit R_7 ableiten, daß $spawn(u_1; u_2 + u_3) \xrightarrow{\checkmark} spawn(u_1; u_2 + u_3)$ gilt. Weiterhin gilt $(spawn(u_1; spawn(u_2) + u_3), spawn(u_1; u_2 + u_3)) \in R$.
  - Für die Annahme $spawn(u_1; u_2 + u_3) \xrightarrow{\omega} t_{new}$ analog.

  Der Beweis für die Teilrelation R_2 erfolgt analog zum Beweis für R_1 mit $u_3 = \mathbf{0}$. Weiterhin können die Techniken aus dem Beweis der Kongruenzeigenschaft von $\sim_\beta$ bzgl. Kanalerzeugung aus Theorem 3.8 verwendet werden.

- Axiom (3.30): $E!F = E'!F'$ falls $E =_\beta E'$ und $F =_\beta F'$.

Sei $R = \{(E!F, E'!F') \mid E =_\beta E', F =_\beta F'\} \cup \{(t,t) \mid t \in \mathcal{P}\}$. Wir zeigen, daß R die Bisimulationsanforderungen aus Definition 3.7 erfüllt. Mit $E =_\beta E'$ und $E : T\ channel$, $E' : T\ channel$ wissen wir, daß $\exists c : [\![E]\!] = c = [\![E']\!]$ gilt.

- Wir nehmen $E!F \xrightarrow{\omega} t_{new}$ an. Dann wissen wir mit R_4, daß $\exists v : [\![F]\!] = v$, $\omega = c!v$ und $t_{new} = \mathbf{1}$ gilt. Aus $F =_\beta F'$ folgt $\exists v', v =_\beta v' : [\![F']\!] = v'$. Dann können wir mit R_4 ableiten, daß $E'!F' \xrightarrow{c!v'} \mathbf{1}$ gilt. Weiterhin gilt $(\mathbf{1}, \mathbf{1}) \in R$.
- Für die Annahme $E'!F' \xrightarrow{\omega} t_{new}$ analog.

- Axiom (3.31): $E?x;t = E'?x;t$ falls $E =_\beta E'$.

  Sei $R = \{(E?x;u, E'?x;u) \mid u \in \mathcal{P}, E =_\beta E'\} \cup \sim_\beta$. Wir zeigen, daß R die Bisimulationsanforderungen aus Definition 3.7 erfüllt.

  - Wir nehmen $E?x;u \xrightarrow{\omega} u_{new}$ an. Dann wissen wir mitR_5, daß $\exists c : [\![E]\!] = c$, $\omega = c?v$ und $u_{new} = u\langle v/x\rangle$ gilt. Aus $E : T\ channel$, $E' : T\ channel$ und $E =_\beta E'$ folgt $[\![E']\!] = c$. Sei $v' =_\beta v$ beliebig. Dann können wir mit R_5 ableiten, daß$E'?x;u \xrightarrow{c?v'} u\langle v'/x\rangle$ gilt. Mit Lemma 3.9 erhalten wir $u\langle v/x\rangle \sim_\beta u\langle v'/x\rangle$. Daher gilt $(u\langle v/x\rangle, u\langle v'/x\rangle) \in R$.
  - Für die Annahme $E'?x;u \xrightarrow{\omega} u_{new}$ analog.

- Axiom (3.32): $\{\vec{x} \leftarrow \vec{E}\};t = \{\vec{x} \leftarrow \vec{E'}\};t$ falls $\vec{E} =_\beta \vec{E'}$.

  Sei $R = \{(\{\vec{x} \leftarrow \vec{E}\};u, \{\vec{x} \leftarrow \vec{E'}\};u) \mid u \in \mathcal{P}, \vec{E} =_\beta \vec{E'}\}$. Wir zeigen, daß R die Bisimulationseigenschaften aus Definition 3.7 erfüllt.

  - Wir nehmen $\{\vec{x} \leftarrow \vec{E}\};u \xrightarrow{\omega} u_{new}$ an. Dann wissen wir mit R_6, daß $\exists \vec{v} : [\![\vec{E}]\!] = \vec{v}$, $\omega = \tau$ und $u_{new} = u\langle \vec{v}/\vec{x}\rangle$ gilt. Aus $\vec{E} =_\beta \vec{E'}$ folgt $\exists \vec{v'}, \vec{v} =_\beta \vec{v'} : [\![\vec{E'}]\!] = \vec{v'}$. Dann können wir mit R_6 ableiten, daß $\{\vec{x} \leftarrow \vec{E'}\};u \xrightarrow{\tau} u\langle \vec{v'}/\vec{x}\rangle$ gilt. Mit Lemma 3.9 erhalten wir $u\langle \vec{v}/\vec{x}\rangle \sim_\beta u\langle \vec{v'}/\vec{x}\rangle$. Daher gilt $(u\langle \vec{v}/\vec{x}\rangle, u\langle \vec{v'}/\vec{x}\rangle) \in R$.
  - Für die Annahme $\{\vec{x} \leftarrow \vec{E'}\};u \xrightarrow{\omega} u_{new}$ analog.

- Axiom (3.33): $[E] = [E']$ falls $E =_\beta E'$.

  Sei $R = \{([E], [E']) \mid E =_\beta E'\} \cup \{(t,t) \mid t \in \mathcal{P}\}$. Wir zeigen, daß R die Bisimulationsanforderungen aus Definition 3.7 erfüllt.

  - Wir nehmen $[E] \xrightarrow{\omega} t_{new}$ an. Dann wissen wir mit R_3, daß $[\![E]\!] = true$, $\omega = \tau$ und $t_{new} = \mathbf{1}$ gilt. Aus $E =_\beta E'$ folgt $[\![E']\!] = true$. Dann können wir mit R_3 ableiten, daß $[E'] \xrightarrow{\tau} \mathbf{1}$ gilt. Weiterhin gilt $(\mathbf{1}, \mathbf{1}) \in R$.
  - Für die Annahme $[E'] \xrightarrow{\omega} t_{new}$ analog.

- Axiom (3.34): $P(\vec{E}) = P(\vec{E'})$ falls $\vec{E} =_\beta \vec{E'}$.

  Sei $R = \{(P(\vec{E}), P(\vec{E'})) \mid \vec{E} =_\beta \vec{E'}\} \cup \sim_\beta$. Wir zeigen, daß R die Bisimulationseigenschaften aus Definition 3.7 erfüllt. Sei $\Theta : P(\vec{x} : \vec{T}) \mapsto t$.

  - Wir nehmen $P(\vec{E}) \xrightarrow{\omega} t_{new}$ an. Dann wissen wir mit R_{11}, daß $\exists \vec{v} : [\![\vec{E}]\!] = \vec{v}$, $\omega = \tau$ und $t_{new} = t\langle\vec{v}/\vec{x}\rangle$ gilt. Aus $\vec{E} =_\beta \vec{E'}$ folgt $\exists \vec{v'}, \vec{v} =_\beta \vec{v'} : [\![\vec{E'}]\!] = \vec{v'}$. Dann können wir mit R_{11} folgern, daß $P(\vec{E'}) \xrightarrow{\tau} t\langle\vec{v'}/\vec{x}\rangle$ gilt. Mit Lemma 3.9 erhalten wir $t\langle\vec{v}/\vec{x}\rangle \sim_\beta t\langle\vec{v'}/\vec{x}\rangle$. Daher gilt $(t\langle\vec{v}/\vec{x}\rangle, t\langle\vec{v'}/\vec{x}\rangle) \in R$.
  - Für die Annahme $P(\vec{E'}) \xrightarrow{\omega} t_{new}$ analog.

- Axiom (3.36): Expansionsgesetz.

  Sei $t_0 = \sum_{i\in\mathcal{I}} \delta_i; t_i + \sum_{j\in\mathcal{J}} E_j?x_j; t_j$, Sei $u_0 = \sum_{k\in\mathcal{K}} \delta_k; u_k + \sum_{l\in\mathcal{L}} E_l?x_l; u_l$. Sei

$$
\begin{array}{lcl}
R_1 & = & \left\{ \begin{array}{l}
(\mathit{spawn}(t_0); u_0, \\
\sum_{i\in\mathcal{I}} \delta_i; \mathit{spawn}(t_i); u_0 + \sum_{k\in\mathcal{K}} \delta_k; \mathit{spawn}(t_0); u_k + \\
\sum_{j\in\mathcal{J}} E_j?x_j; \mathit{spawn}(t_j); u_0 + \sum_{l\in\mathcal{L}} E_l?x_l; \mathit{spawn}(t_0); u_l + \\
\sum_{\delta_i=(c!v,C), [\![E_l]\!]=c}^{i\in\mathcal{I}, l\in cL} \tau; \mathbf{ch}\ C.\ \mathit{spawn}(t_i); u_l\langle v/x_l\rangle + \\
\sum_{[\![E_j]\!]=c, \delta_k=c!v,C}^{j\in\mathcal{J}, k\in\mathcal{K}} \tau; \mathbf{ch}\ C.\ \mathit{spawn}(t_j\langle v/x_j\rangle); u_k) \mid \\
\forall j \in \mathcal{J} : x_j \notin \mathit{fv}(u_0), \forall l \in \mathcal{L} : x_l \notin \mathit{fv}(t)
\end{array} \right\} \\
R_2 & = & \{(\mathbf{ch}\ C.\ \mathit{spawn}(\mathbf{1}; t'); u', \mathbf{1}; \mathbf{ch}\ C.\ \mathit{spawn}(t'); u') \mid t', u' \in \mathcal{P}\} \\
R_3 & = & \{(\mathbf{ch}\ C.\ \mathit{spawn}(t'); \mathbf{1}; u', \mathbf{1}; \mathbf{ch}\ C.\ \mathit{spawn}(t'); u') \mid t', u' \in \mathcal{P}\} \\
R_4 & = & \{(t', t') \mid t' \in \mathcal{P}\}
\end{array}
$$

  Sei $R = R_1 \cup R_2 \cup R_3 \cup R_4$. Wir zeigen, daß R die Bisimulationsanforderungen aus Definition 3.7 erfüllt. Wir bezeichnen die linken Seiten der Paare in R mit t_L und die rechten Seiten mit t_R. Zunächst betrachten wir die Teilrelation R_1.

  - Wir nehmen $t_L \xrightarrow{\omega} t_{new}$ an und unterscheiden die folgenden Fälle:
    - $\exists i \in \mathcal{I} : \delta_i \xrightarrow{\delta_i} \mathbf{1}$, $\omega = \delta_i$ und $t_{new} = \mathit{spawn}(\mathbf{1}; t_i); u_0$, abgeleitet mit R_{12}, R_8 und den Regeln für Auswahl. Für $\delta_i = [E]$ ist $\omega = \tau$. Dann können wir mit R_{12} und den Regeln für Auswahl schließen, daß $t_R \xrightarrow{\delta_i} \mathbf{1}; \mathit{spawn}(t_i); u_0$ gilt. Weiterhin gilt $(\mathit{spawn}(\mathbf{1}; t_i); u_0, \mathbf{1}; \mathit{spawn}(t_i); u_0) \in R$.
    - $\exists k \in \mathcal{K} : \delta_k \xrightarrow{\delta_k} \mathbf{1}$, $\omega = \delta_k$ und $t_{new} = \mathit{spawn}(t_0); \mathbf{1}; u_k$, abgeleitet mit R_7, R_{12}, R_{13} und den Regeln für Auswahl. Für $\delta_k = [E]$ ist $\omega = \tau$. Dann können wir mit R_{12} und den Regeln für Auswahl schließen, daß $t_R \xrightarrow{\omega} \mathbf{1}; \mathit{spawn}(t_0); u_k$ gilt. Weiterhin gilt $(\mathit{spawn}(t_0); \mathbf{1}; u_k, \mathbf{1}; \mathit{spawn}(t_0); u_k) \in R$.

* $\exists j \in \mathcal{J} : E_j?x_j; t_j \xrightarrow{c?v} t_j\langle v/x_j\rangle$, $\omega = c?v$ und $t_{new} = spawn(t_j\langle v/x_j\rangle); u_0$, abgeleitet mit R_5, R_8, R_{12} und den Regeln für Auswahl. Mit R_5 und den Regeln für Auswahl wissen wir, daß $t_R \xrightarrow{c?v} (spawn(t_j); u_0)\langle v/x_j\rangle$ gilt. Aus $x_j \notin fv(u_0)$ folgt $(spawn(t_j); u_0)\langle v/x_j\rangle = spawn(t_j\langle v/x_j\rangle); u_0$. Weiterhin gilt $(spawn(t_j\langle v/x_j\rangle); u_0, spawn(t_j\langle v/x_j\rangle); u_0) \in R$.
* $\exists l \in \mathcal{L} : E_l?x_l; u_l \xrightarrow{c?v} u_l\langle v/x_l\rangle$, $\omega = c?v$ und $t_{new} = spawn(t_0); u_l\langle v/x_l\rangle$, abgeleitet mit R_7, R_5, R_{13} und den Auswahl-Regeln. Dann können wir mit R_5 und den Auswahl-Regeln folgern, daß $t_R \xrightarrow{c?v} (spawn(t_0); u_l)\langle v/x_l\rangle$ gilt. Aus $x_l \notin fv(t_0)$ folgt $(spawn(t_0); u_l)\langle v/x_l\rangle = spawn(t_0); u_l\langle v/x_l\rangle$. Weiterhin gilt $(spawn(t_0); u_l\langle v/x_l\rangle, spawn(t_0); u_l\langle v/x_l\rangle) \in R$.
* $\exists i \in \mathcal{I} : \delta_i \xrightarrow{c!v,C} \mathbf{1}$, $\exists l \in \mathcal{L} : E_l?x_l; u_l \xrightarrow{c?v} u_l\langle v/x_l\rangle$, $\omega = \tau$ und $t_{new} = \mathbf{ch}\ C.\ spawn(\mathbf{1}; t_i); u_l\langle v/x_l\rangle$, abgeleitet mit R_{14} und den Regeln für Auswahl. Dann können wir mit R_2, R_{12} und den Regeln für Auswahl folgern, daß $t_R \xrightarrow{\tau} \mathbf{1}; \mathbf{ch}\ C.\ spawn(t_i); u_l\langle v/x_l\rangle$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\ spawn(\mathbf{1}; t_i); u_l\langle v/x_l\rangle, \mathbf{1}; \mathbf{ch}\ C.\ spawn(t_i); u_l\langle v/x_l\rangle) \in R$.
* $\exists j \in \mathcal{J} : E_j?x_j; t_j \xrightarrow{c?v} t_j\langle v/x_j\rangle$, $\exists k \in \mathcal{K} : \delta_k \xrightarrow{c!v,C} \mathbf{1}$, $\omega = \tau$ und $t_{new} = \mathbf{ch}\ C.\ spawn(t_j\langle v/x_j\rangle); \mathbf{1}; u_k$, abgeleitet mit R_{14} und den Regeln für Auswahl. Dann können wir mit R_2, R_{12} und den Regeln für Auswahl schließen, daß $t_R \xrightarrow{\tau} \mathbf{1}; \mathbf{ch}\ C.\ spawn(t_j\langle v/x_j\rangle); u_k$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\ spawn(t_j\langle v/x_j\rangle); \mathbf{1}; u_k, \mathbf{1}; \mathbf{ch}\ C.\ spawn(t_j\langle v/x_j\rangle); u_k) \in R$.

– Wir nehmen $t_R \xrightarrow{\omega} t_{new}$ an und unterscheiden die folgenden Fälle:

* $\exists i \in \mathcal{I} : \delta_i \xrightarrow{\delta_i} \mathbf{1}$, $\omega = \delta_i$ und $t_{new} = \mathbf{1}; spawn(t_i); u_0$, abgeleitet mit R_{12} und den Regeln für Auswahl. Für $\delta_i = [E]$ ist $\omega = \tau$. Dann können wir mit R_{12}, R_8 und den Regeln für Auswahl schließen, daß $t_L \xrightarrow{\omega} spawn(\mathbf{1}; t_i); u_0$ gilt. Weiterhin gilt $(spawn(\mathbf{1}; t_i); u_0, \mathbf{1}; spawn(t_i); u_0) \in R$.
* $\exists k \in \mathcal{K} : \delta_k \xrightarrow{\delta_k} \mathbf{1}$, $\omega = \delta_k$ und $t_{new} = \mathbf{1}; spawn(t_0); u_k$, abgeleitet mit R_{12} und den Regeln für Auswahl. Für $\delta_k = [E]$ ist $\omega = \tau$. Dann können wir mit R_7, R_{12}, R_{13} und den Auswahl-Regeln schließen, daß $t_L \xrightarrow{\omega} spawn(t_0); \mathbf{1}; u_k$ gilt. Weiterhin gilt $(spawn(t_0); \mathbf{1}; u_k, \mathbf{1}; spawn(t_0); u_k) \in R$.
* $\exists j \in \mathcal{J} : E_j?x_j; t_j \xrightarrow{c?v} t_j\langle v/x_j\rangle$, $\omega = c?v$ und $t_{new} = (spawn(t_j); u_0)\langle v/x_j\rangle$, abgeleitet mit R_5 und den Regeln für Auswahl. Wegen $x_j \notin fv(u_0)$ gilt $(spawn(t_j); u_0)\langle v/x_j\rangle = spawn(t_j\langle v/x_j\rangle); u_0$. Mit R_5, R_8, R_{12} und den Regeln für Auswahl können wir ableiten, daß $t_L \xrightarrow{c?v} spawn(t_j\langle v/x_j\rangle); u_0$ gilt. Weiterhin gilt $(spawn(t_j\langle v/x_j\rangle); u_0, spawn(t_j\langle v/x_j\rangle); u_0) \in R$.

* $\exists l \in \mathcal{L} : E_l?x_l; u_l \xrightarrow{c?v} u_l\langle v/x_l\rangle$, $\omega = c?v$ und $t_{new} = (spawn(t_0); u_l)\langle v/x_l\rangle$, abgeleitet mit R_5 und den Regeln für Auswahl. Wegen $x_l \notin fv(t_0)$ gilt $(spawn(t_0); u_l)\langle v/x_l\rangle = spawn(t_0); u_l\langle v/x_l\rangle$. Mit R_5, R_7, R_{13} und den Regeln für Auswahl können wir ableiten, daß $t_L \xrightarrow{c?v} spawn(t_0); u_l\langle v/x_l\rangle$ gilt. Weiterhin gilt $(spawn(t_0); u_l\langle v/x_l\rangle, spawn(t_0); u_l\langle v/x_l\rangle) \in R$.
* $\exists i \in \mathcal{I} : \delta_i \xrightarrow{c!v,C} \mathbf{1}$, $\exists l \in \mathcal{L} : E_l?x_l; u_l \xrightarrow{c?v} u_l\langle v/x_l\rangle$, $\omega = \tau$ und $t_{new} = \mathbf{1}; \mathbf{ch}\ C.\, spawn(t_i); u_l\langle v/x_l\rangle$, abgeleitet mit R_2, R_{12} und den Regeln für Auswahl. Dann können wir mit R_{14} und den Regeln für Auswahl ableiten, daß $t_L \xrightarrow{\tau} \mathbf{ch}\ C.\, spawn(\mathbf{1}; t_i); u_l\langle v/x_l\rangle$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\, spawn(\mathbf{1}; t_i); u_l\langle v/x_l\rangle, \mathbf{1}; \mathbf{ch}\ C.\, spawn(t_i); u_l\langle v/x_l\rangle) \in R$.
* $\exists j \in \mathcal{J} : E_j?x_j; t_j \xrightarrow{c?v} t_j\langle v/x_j\rangle$, $\exists k \in \mathcal{K} : \delta_k \xrightarrow{c!v,C} \mathbf{1}$, $\omega = \tau$ und $t_{new} = \mathbf{1}; \mathbf{ch}\ C.\, spawn(t_j\langle v/x_j\rangle); u_k$, abgeleitet mit R_2, R_{12} und den Regeln für Auswahl. Dann können wir mit R_{14} und den Regeln für Auswahl schließen, daß $t_L \xrightarrow{\tau} \mathbf{ch}\ C.\, spawn(t_j\langle v/x_j\rangle); \mathbf{1}; u_k$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\, spawn(t_j\langle v/x_j\rangle); \mathbf{1}; u_k, \mathbf{1}; \mathbf{ch}\ C.\, spawn(t_j\langle v/x_j\rangle); u_k) \in R$.

Für R_2 und R_3 können wir die Axiome (3.3), (3.4) und die Kongruenz von $\sim_\beta$ bzgl. Kanalerzeugung verwenden.

□

Korollar 3.11. Folgende Gleichungen sind Beispiele für aus $\mathcal{AX}_{\sim_\beta}$ abgeleitete Gleichungen:

$$\begin{aligned} spawn(spawn(t)) &= spawn(t), \\ spawn(\mathbf{1}) &= \mathbf{1}, \\ spawn(t_1; spawn(t_2)) &= spawn(t_1; t_2). \end{aligned}$$

Beweis für Korollar 3.11. Wir beweisen die Gültigkeit der Gleichungen durch Umformung des jeweils linken Terms in den jeweils rechten durch Anwendung der Axiome aus Abbildung 3.4.

1. $spawn(spawn(t)) = t$:

$$\begin{aligned} & spawn(spawn(t)) & \\ = \ & spawn(\mathbf{1}; spawn(t)) & (Axiom\ (3.3)) \\ = \ & spawn(\mathbf{1}; spawn(t) + \mathbf{0}) & (Axiom\ (3.8)) \\ = \ & spawn(\mathbf{1}; t + \mathbf{0}) & (Axiom\ (3.29)) \\ = \ & spawn(\mathbf{1}; t) & (Axiom\ (3.8)) \\ = \ & spawn(t) & (Axiom\ (3.3)) \end{aligned}$$

2. $spawn(\mathbf{1}) = \mathbf{1}$:

$$\begin{array}{lll} & spawn(\mathbf{1}) & \\ = & spawn(spawn(\mathbf{0})) & (Axiom\ (3.26)) \\ = & spawn(\mathbf{0}) & (Gleichung\ 1) \\ = & \mathbf{1} & (Axiom\ (3.26)) \end{array}$$

3. $spawn(t_1; spawn(t_2)) = spawn(t_1; t_2)$:

$$\begin{array}{lll} & spawn(t_1; spawn(t_2)) & \\ = & spawn(t_1; spawn(t_2) + \mathbf{0}) & (Axiom\ (3.8)) \\ = & spawn(t_1; t_2 + \mathbf{0}) & (Axiom\ (3.29)) \\ = & spawn(t_1; t_2) & (Axiom\ (3.3)) \end{array}$$

□

Theorem 3.13. $\approx_\beta$ ist eine Kongruenz bzgl. *spawn*, Definition von Bezeichnern, Eingabeaktionen, Kanalrestriktion und sequentieller Komposition.

Wir zeigen die Kongruenzeigenschaft von $\approx_\beta$ bzgl. *spawn*, Deklaration von Bezeichnern, Eingabeaktionen, Kanalrestriktion und sequentieller Komposition. Für den Beweis benötigen wir das folgende Lemma:

Lemma 3.20 *Mit $\sigma =_\beta \sigma'$ gilt $t\sigma \approx_\beta t\sigma'$.*

Beweis für Lemma 3.20. Folgt direkt aus Lemma 3.9 und $\sim_\beta \subset \approx_\beta$. □

Beweis für Theorem 3.13. Für jeden Transitionsschritt $t \xrightarrow{c!v,C} t'$ nehmen wir, wie in Abbildung 3.3 gefordert, die Gültigkeit der Bedingung $C \cap fc(t) = \emptyset$ an.

- $spawn(t) \approx_\beta spawn(u)$ falls $t \approx_\beta u$.

  Sei $R = \{(spawn(t_0), spawn(u_0)) \mid t_0 \approx_\beta u_0\}$. Wir zeigen, daß R die Anforderungen aus Definition 3.12 erfüllt. Wegen der Symmetrie von R zeigen wir den Beweis nur für eine Richtung.

  Wir nehmen $spawn(t_0) \xrightarrow{\omega} t'$ an und unterscheiden die folgenden Fälle:

  - $t_0 \xrightarrow{c?v} t_0'$, $\omega = c?v$ und $t' = spawn(t_0')$, abgeleitet mit R_8. Aus $t_0 \approx_\beta u_0$ folgt $\forall v', v =_\beta v', \exists u_0' : u_0 \xRightarrow{\widehat{c?v'}} u_0'$ und $t_0' \approx_\beta u_0'$. Dann können wir mit der wiederholten Anwendung von R_8 ableiten, daß $spawn(u_0) \xRightarrow{\widehat{c?v'}} spawn(u_0')$ gilt. Weiterhin gilt $(spawn(t_0'), spawn(u_0')) \in R$.
  - $t_0 \xrightarrow{\alpha} t_0'$, $\alpha = \tau$ oder $\alpha = (c!v, C)$, $\omega = \alpha$ und $t' = spawn(t_0')$, abgeleitet mit R_8. Aus $t_0 \approx_\beta u_0$ folgt $\exists \alpha', \alpha =_\beta \alpha', \exists u_0' : u_0 \xRightarrow{\widehat{\alpha'}} u_0'$ und $t_0' \approx_\beta u_0'$. Dann können wir mit der wiederholten Anwendung von R_8 schließen, daß $spawn(u_0) \xRightarrow{\widehat{\alpha'}} spawn(u_0')$ gilt. Weiterhin gilt $(spawn(t_0'), spawn(u_0')) \in R$.

– $\omega = \checkmark$ and $t' = spawn(t_0)$, abgeleitet mit R_7. Ebenso können wir mit R_7 ableiten, daß $spawn(u_0) \xrightarrow{\checkmark} spawn(u_0)$ und daher auch $spawn(u_0) \overset{\widehat{\checkmark}}{\Longrightarrow} spawn(u_0)$ gilt. Weiterhin gilt $(spawn(t_0), spawn(u_0)) \in R$.

- $\{\vec{x} \leftarrow \vec{E}\}; t \approx_\beta \{\vec{x} \leftarrow \vec{E}'\}; u$ falls $t \approx_\beta u$ und $\vec{E} =_\beta \vec{E}'$.

  Sei $R = \{(\{\vec{x} \leftarrow \vec{E}\}; t_0, \{\vec{x} \leftarrow \vec{E}'\}; u_0) \mid t_0 \approx_\beta u_0, \vec{E} =_\beta \vec{E}'\} \cup \approx_\beta$. Wir zeigen, daß R die Anforderungen aus Definition 3.12 erfüllt. Wegen der Symmetrie von R zeigen wir den Beweis nur für eine Richtung.

  Wir nehmen $\{\vec{x} \leftarrow \vec{E}\}; t_0 \xrightarrow{\omega} t'$ an und unterscheiden die folgenden Fälle:

  – $[\![\vec{E}]\!] = \vec{v}$, $t_0\langle \vec{v}/\vec{x} \rangle \xrightarrow{c?v} t_0'$ und $t' = t_0'$, abgeleitet mit R_6. Sei $\vec{v}' = [\![\vec{E}']\!]$. Mit $t_0 \approx_\beta u_0$ und Lemma 3.20 können wir folgern, daß $t_0\langle \vec{v}/\vec{x} \rangle \approx_\beta u_0\langle \vec{v}'/\vec{x} \rangle$ gilt. Dann wissen wir, daß $\forall v', v =_\beta v', \exists u_0' : u_0\langle \vec{v}'/\vec{x} \rangle \overset{\widehat{c?v'}}{\Longrightarrow} u_0'$ und $t_0' \approx_\beta u_0'$ gilt. Daher können wir mit R_6 ableiten, daß $\{\vec{x} \leftarrow \vec{E}'\}; u_0 \overset{\widehat{c?v'}}{\Longrightarrow} u_0'$ gilt. Weiterhin gilt $(t_0', u_0') \in R$.

  – $[\![\vec{E}]\!] = \vec{v}$, $t_0\langle \vec{v}/\vec{x} \rangle \xrightarrow{\omega} t_0'$, $\omega \neq c?v$ und $t' = t_0'$, abgeleitet mit R_6. Sei $\vec{v}' = [\![\vec{E}']\!]$. Mit $t_0 \approx_\beta u_0$ und Lemma 3.20 wissen wir, daß $t_0\langle \vec{v}/\vec{x} \rangle \approx_\beta u_0\langle \vec{v}'/\vec{x} \rangle$ gilt. Dann wissen wir auch, daß $\exists \omega', \omega =_\beta \omega', \exists u_0' : u_0\langle \vec{v}'/\vec{x} \rangle \overset{\widehat{\omega'}}{\Longrightarrow} u_0'$ und $t_0' \approx_\beta u_0'$ gilt. Mit R_6 können wir $\{\vec{x} \leftarrow \vec{E}'\}; u_0 \overset{\widehat{\omega'}}{\Longrightarrow} u_0'$ folgern. Weiterhin gilt $(t_0', u_0') \in R$.

- $E?x; t \approx_\beta E'?x; u$ falls $t \approx_\beta u$ und $E =_\beta E'$.

  Sei $R = \{(E?x; t_0, E'?x; u_0) \mid t_0 \approx_\beta u_0, E =_\beta E'\} \cup \approx_\beta$. Wir zeigen, daß R die Anforderungen aus Definition 3.12 erfüllt. Wegen der Symmetrie von R zeigen wir den Beweis nur für eine Richtung.

  Wir nehmen $E?x; t_0 \xrightarrow{\omega} t'$ an. Für Kanäle entspricht β-Äquivalenz der Identität, daher $\exists c : [\![E]\!] = c = [\![E']\!]$. Dann wissen wir mit R_5, daß $\omega = c?v$ und $t' = t_0\langle v/x \rangle$ gilt. Sei $v' =_\beta v$ beliebig. Dann können wir mit R_5 ableiten, daß $E'?x; u_0 \xrightarrow{c?v'} u_0\langle v'/x \rangle$ gilt. Daher gilt auch $E'?x; u_0 \overset{\widehat{c?v'}}{\Longrightarrow} u_0\langle v'/x \rangle$. Aus $t_0 \approx_\beta u_0$ folgt $\forall \sigma : t_0\sigma \approx_\beta u_0\sigma$. Daher können wir mit Lemma 3.20 ableiten, daß $t_0\langle v/x \rangle \approx_\beta u_0\langle v'/x \rangle$ gilt. Weiterhin gilt $(t_0\langle v/x \rangle, u_0\langle v'/x \rangle) \in R$.

- **ch** $C.\, t \approx_\beta$ **ch** $C.\, u$ falls $t \approx_\beta u$.

  Sei $\{(\textbf{ch}\ D.\, t_0, \textbf{ch}\ D.\, u_0) \mid t_0 \approx_\beta u_0\}$. Wir zeigen, daß R die Anforderungen aus Definition 3.12 erfüllt. Wegen der Symmetrie von R zeigen wir den Beweis nur für eine Richtung.

  Wir nehmen **ch** $C.\, t_0 \xrightarrow{\omega} t'$ an und unterscheiden die folgenden Fälle:

- $t_0 \xrightarrow{c?v} t_0'$, $\omega = c?v$, $c \notin C$, $fv(c) \cap C = \emptyset$ und $t' = \mathbf{ch}\ C.\ t_0'$, abgeleitet mit R_{15}. Aus $t_0 \approx_\beta u_0$ folgt $\forall v', v =_\beta v', \exists u_0' : u_0 \xRightarrow{\widehat{c?v'}} u_0'$ und $t_0' \approx_\beta u_0'$. Weiterhin wissen wir mit $v =_\beta v'$, daß $fc(v) = fc(v')$ gilt. Daher können wir $fc(v') \cap C = \emptyset$ folgern. Dann wissen wir mit der wiederholten Anwendung von R_{15}, daß $\mathbf{ch}\ C.\ u_0 \xRightarrow{\widehat{c?v'}} \mathbf{ch}\ C.\ u_0'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\ t_0', \mathbf{ch}\ C.\ u_0') \in R$.
- $t_0 \xrightarrow{\omega} t_0'$, $\omega \neq c?v$, $fc(\omega) \cap C = \emptyset$ und $t' = \mathbf{ch}\ C.\ t_0'$, abgeleitet mit R_{15}. Aus $t_0 \approx_\beta u_0$ folgt $\exists \omega', \omega =_\beta \omega', \exists u_0' : u_0 \xRightarrow{\widehat{\omega'}} u_0'$ und $t_0' \approx_\beta u_0'$. Aus $\omega =_\beta \omega'$ folgt $fc(\omega) = fc(\omega')$; daher gilt $fc(\omega') \cap C = \emptyset$. Dann können wir mit der wiederholten Anwendung von R_{15} schließen, daß $\mathbf{ch}\ C.\ u_0 \xRightarrow{\widehat{\omega'}} \mathbf{ch}\ C.\ u_0'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\ t_0', \mathbf{ch}\ C.\ u_0') \in R$.
- $t_0 \xrightarrow{c!v,A} t_0'$, $A \cap C = \emptyset$, $fc(v) \cap C = \emptyset$, $\omega = (c!v, A \cup (C \cap fc(v)))$ und $t' = \mathbf{ch}\ C \setminus fc(v).\ t_0'$, abgeleitet mit R_{16}. Aus $t_0 \approx_\beta u_0$ folgt $\exists v', v =_\beta v', \exists u_0' : u_0 \xRightarrow{\widehat{c!v',A}} u_0'$ und $t_0' \approx_\beta u_0'$. Aus $v =_\beta v'$ folgt $fc(v) = fc(v')$; daher ist $fc(v') \cap C = fc(v) \cap C \neq \emptyset$. Dann können wir mit R_{16} und der wiederholten Anwendung von R_{15} folgern, daß $\mathbf{ch}\ C.u_0 \xRightarrow{\widehat{c!v,A \cup (C \cap fc(v'))}} \mathbf{ch}\ C \setminus fc(v').u_0'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C \setminus fc(v).t_0', \mathbf{ch}\ C \setminus fc(v').u_0') \in R$.

- $t_1; t_2 \approx_\beta u_1; u_2$ falls $t_1 \approx_\beta u_1$ und $t_2 \approx_\beta u_2$.

  Sei $R = \{(\mathbf{ch}\ D.\ t_1; t_2, \mathbf{ch}\ D.\ u_1; u_2) \mid t_1 \approx_\beta u_1, t_2 \approx_\beta u_2\}$. Wir zeigen, daß R die Anforderungen aus Definition 3.12 erfüllt. Wegen der Symmetrie von R zeigen wir den Beweis nur für eine Richtung.

  Wir beweisen den Fall $D = \emptyset$. Für $D \neq \emptyset$ können die Techniken aus dem obigen Beweis der Kongruenz von $\approx_\beta$ bzgl. Kanalerzeugung verwendet werden. Wir nehmen $t_1; t_2 \xrightarrow{\omega} t_{new}$ an und unterscheiden die folgenden Fälle:

  - $t_1 \xrightarrow{c?v} t_1'$, $\omega = c?v$ und $t_{new} = t_1'; t_2$, abgeleitet mit R_{12}. Aus $t_1 \approx_\beta u_1$ folgt $\forall v', v =_\beta v', \exists u_1' : u_1 \xRightarrow{\widehat{c?v'}} u_1'$ und $t_1' \approx_\beta u_1'$. Dann können wir mit der wiederholten Anwendung von R_{12} folgern, daß $u_1; u_2 \xRightarrow{\widehat{c?v'}} u_1'; u_2$ gilt. Weiterhin gilt $(t_1'; t_2, u_1'; u_2) \in R$.
  - $t_1 \xrightarrow{\alpha} t_1'$, $\alpha \neq c?v$ und $t_{new} = t_1'; t_2$, abgeleitet mit R_{12}. Aus $t_1 \approx_\beta u_1$ folgt $\exists \alpha', \alpha =_\beta \alpha', \exists u_1' : u_1 \xRightarrow{\widehat{\alpha'}} u_1'$ und $t_1' \approx_\beta u_1'$. Dann können wir mit der wiederholten Anwendung von R_{12} folgern, daß $u_1; u_2 \xRightarrow{\widehat{\alpha'}} u_1'; u_2$ gilt. Weiterhin gilt $(t_1'; t_2, u_1'; u_2) \in R$.
  - $t_1 \xrightarrow{\checkmark} t_1'$, $t_2 \xrightarrow{c?v} t_2'$, $\omega = c?v$ und $t_{new} = t_1'; t_2'$, abgeleitet mit R_{13}. Aus $t_1 \approx_\beta u_1$ folgt $\exists u_1' : u_1 \xRightarrow{\widehat{\checkmark}} u_1'$ und $t_1' \approx_\beta u_1'$. Aus $t_2 \approx_\beta u_2$ folgt

$\forall v', v =_\beta v', \exists u_2' : u_2 \stackrel{\widehat{c?v'}}{\Longrightarrow} u_2'$ und $t_2' \approx_\beta u_2'$. Dann können wir mit der wiederholten Anwendung von R_{12} und R_{13} folgern, daß $u_1; u_2 \stackrel{\widehat{c?v'}}{\Longrightarrow} u_1'; u_2'$ gilt. Weiterhin gilt $(t_1'; t_2', u_1'; u_2') \in R$.

- $t_1 \xrightarrow{\checkmark} t_1'$, $t_2 \xrightarrow{\omega} t_2'$, $\omega \neq c?v$ und $t_{new} = t_1'; t_2'$, abgeleitet mit R_{13}. Aus $t_1 \approx_\beta u_1$ folgt $\exists u_1' : u_1 \stackrel{\widehat{\checkmark}}{\Longrightarrow} u_1'$ und $t_1' \approx_\beta u_1'$. Aus $t_2 \approx_\beta u_2$ folgt $\exists \omega', \omega =_\beta \omega', \exists u_2' : u_2 \stackrel{\widehat{\omega'}}{\Longrightarrow} u_2'$ und $t_2' \approx_\beta u_2'$. Dann können wir mit der wiederholten Anwendung von R_{12} und R_{13} folgern, daß $u_1; u_2 \stackrel{\widehat{\omega'}}{\Longrightarrow} u_1'; u_2'$ gilt. Weiterhin gilt $(t_1'; t_2', u_1'; u_2') \in R$.
- $t_1 \xrightarrow{\checkmark} t_1'$, $t_1' \xrightarrow{c?v} t_1''$, $t_2 \xrightarrow{c!v,C} t_2'$, $\omega = \tau$ und $t_{new} = \mathbf{ch}\ C.\ t_1''; t_2'$, abgeleitet mit R_{14}. Aus $t_2 \approx_\beta u_2$ folgt $\exists v', v =_\beta v', \exists u_2' : u_2 \stackrel{\widehat{c!v',C}}{\Longrightarrow} u_2'$ und $t_2' \approx_\beta u_2'$. Aus $t_1 \approx_\beta u_1$ folgt $\exists u_1' : u_1 \stackrel{\widehat{\checkmark}}{\Longrightarrow} u_1'$, $t_1' \approx_\beta u_1'$ und $\forall v'', v =_\beta v'', \exists u_1'' : u_1' \stackrel{\widehat{c?v''}}{\Longrightarrow} u_1''$, $t_1'' \approx_\beta u_1''$. Dann können wir mit $v' =_\beta v''$ ableiten, daß $\exists u_v : u_1' \stackrel{\widehat{c?v'}}{\Longrightarrow} u_v$ und $t_1'' \approx_\beta u_v$ gilt. Dann können wir mit R_{14} und der wiederholten Anwendung von R_{12} und R_{13} folgern, daß $u_1; u_2 \stackrel{\tau}{\Longrightarrow} \mathbf{ch}\ C.\ u_v; u_2'$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\ t_1''; t_2', \mathbf{ch}\ C.\ u_v; u_2') \in R$.
- $t_1 \xrightarrow{\checkmark} t_1'$, $t_1' \xrightarrow{c!v,C} t_1''$, $t_2 \xrightarrow{c?v} t_2'$, $\omega = \tau$ und $t_{new} = \mathbf{ch}\ C.\ t_1''; t_2'$, abgeleitet mit R_{14}. Aus $t_1 \approx_\beta u_1$ folgt $\exists u_1' : u_1 \stackrel{\widehat{\checkmark}}{\Longrightarrow} u_1'$, $t_1' \approx_\beta u_1'$ und $\exists v', v =_\beta v', \exists u_1'' : u_1' \stackrel{\widehat{c!v',C}}{\Longrightarrow} u_1''$, $t_1'' \approx_\beta u_1''$. Aus $t_2 \approx_\beta u_2$ folgt $\forall v'', v =_\beta v'', \exists u_2' : u_2 \stackrel{\widehat{c?v''}}{\Longrightarrow} u_2'$ und $t_2' \approx_\beta u_2'$. Dann können wir mit $v' =_\beta v''$ schließen, daß $\exists u_v : u_2 \stackrel{\widehat{c?v'}}{\Longrightarrow} u_v$ und $t_2' \approx_\beta u_v$ gilt. Daher können wir mit R_{14} und der wiederholten Anwendung von R_{12} und R_{13} folgern, daß $u_1; u_2 \stackrel{\tau}{\Longrightarrow} \mathbf{ch}\ C.\ u_1''; u_v$ gilt. Weiterhin gilt $(\mathbf{ch}\ C.\ t_1''; t_2', \mathbf{ch}\ C.\ u_1''; u_v) \in R$.

□

Proposition 3.14. Für alle $t \in \mathcal{P}$ gilt: $\tau; t \approx_\beta t$ und $spawn(\tau; t) \approx_\beta spawn(t)$. Weiterhin gilt $\mathbf{1}; \tau \approx_\beta \mathbf{1}$.

Beweis für Proposition 3.14.

- $\tau; t \approx_\beta t$.

  Sei $R = \{(\tau; u, u) \mid u \in \mathcal{P}\} \cup \sim_\beta$. Wir zeigen, daß R die Anforderungen aus Definition 3.12 erfüllt. Mit Theorem 3.10 wissen wir, daß $\forall u \in \mathcal{P} : \mathbf{1}; u \sim_\beta u$ gilt.

- Wir nehmen $\tau; u \xrightarrow{\omega} u_{new}$ an. Dann ist $\omega = \tau$ und $u_{new} = \mathbf{1}; u$, abgeleitet mit R_2, R_{12}. Mit der Definition von $\Rightarrow$ wissen wir, daß $u \stackrel{\hat{\tau}}{\Longrightarrow} u$ gilt. Weiterhin wissen wir mit $\mathbf{1}; u \sim_\beta u$, daß $(\mathbf{1}; u, u) \in R$ gilt.
- Wir nehmen $u \xrightarrow{\omega} u'$ an. Mit R_2, R_{12} können wir ableiten, daß $\tau; u \xrightarrow{\tau} \mathbf{1}; u$ gilt. Aus $u \sim_\beta \mathbf{1}; u$ folgt $(\mathbf{1}; u, u) \in R$. Weiterhin können wir mit R_{13} ableiten, daß $\mathbf{1}; u \xrightarrow{\omega} \mathbf{1}; u'$ und $(\mathbf{1}; u', u') \in R$ gilt. Daher gilt $\tau; u \stackrel{\hat{\omega}}{\Longrightarrow} \mathbf{1}; u'$ und $(\mathbf{1}; u', u') \in R$.

- $spawn(\tau; t) \approx_\beta spawn(t)$.

  Sei $R = \{(spawn(\tau; u), spawn(u)) \mid u \in \mathcal{P}\} \cup \sim_\beta$. Wir zeigen, daß R die Anforderungen aus Definition 3.12 erfüllt. Mit Theorem 3.10 wissen wir, daß $\forall u \in \mathcal{P} : spawn(\mathbf{1}; u) \sim_\beta spawn(u)$ gilt.

  - Wir nehmen $spawn(\tau; u) \xrightarrow{\omega} u_{new}$ an und unterscheiden die folgenden Fälle:
    - $\tau; u \xrightarrow{\tau} \mathbf{1}; u$. Dann ist $\omega = \tau$ und $u_{new} = spawn(\mathbf{1}; u)$, abgeleitet mit R_2, R_{12}, R_8. Mit der Definition von $\Rightarrow$ wissen wir $u \stackrel{\hat{\tau}}{\Longrightarrow} u$ und daher auch $spawn(u) \stackrel{\hat{\tau}}{\Longrightarrow} spawn(u)$. Weiterhin wissen wir mit $spawn(\mathbf{1}; u) \sim_\beta spawn(u)$, daß $(spawn(\mathbf{1}; u), spawn(u)) \in R$.
    - $\omega = \checkmark$ und $u_{new} = spawn(\tau; u)$, abgeleitet mit R_7. Ebenso können wir mit R_7 ableiten, daß $spawn(u) \xrightarrow{\checkmark} spawn(u)$ und damit auch $spawn(u) \stackrel{\widehat{\checkmark}}{\Longrightarrow} spawn(u)$ gilt. Weiterhin gilt $(spawn(\tau; u), spawn(u)) \in R$.
  - Wir nehmen $spawn(u) \xrightarrow{\omega} u_{new}$ an und unterscheiden die folgenden Fälle:
    - $u \xrightarrow{\alpha} u'$, $\omega = \alpha$ und $u_{new} = spawn(u')$, abgeleitet mit R_8. Mit R_2, R_{12}, R_8 können wir $spawn(\tau; u) \xrightarrow{\tau} spawn(\mathbf{1}; u)$ folgern. Aus $spawn(\mathbf{1}; u) \sim_\beta spawn(u)$ folgt $(spawn(\mathbf{1}; u), spawn(u)) \in R$. Weiterhin können wir mit R_{13} ableiten, daß $spawn(\mathbf{1}; u) \xrightarrow{\omega} spawn(\mathbf{1}; u')$ und $(spawn(\mathbf{1}; u'), spawn(u')) \in R$ gilt. Daher gilt $spawn(\tau; u) \stackrel{\hat{\omega}}{\Longrightarrow} spawn(\mathbf{1}; u')$ und $spawn(\mathbf{1}; u', u') \in R$.
    - $\omega = \checkmark$ und $u_{new} = spawn(u)$, abgeleitet mit R_7. Ebenso können wir mit R_7 ableiten, daß $spawn(\tau; u) \xrightarrow{\checkmark} spawn(\tau; u)$ und daher auch $spawn(\tau; u) \stackrel{\widehat{\checkmark}}{\Longrightarrow} spawn(\tau; u)$ gilt. Weiterhin gilt $(spawn(\tau; u), spawn(u)) \in R$.

- $\mathbf{1}; \tau \approx_\beta \mathbf{1}$.

  Sei $R = \{(\mathbf{1}; \tau, \mathbf{1})\} \cup \sim_\beta$. Wir zeigen, daß R die Anforderungen aus Definition 3.12 erfüllt.

– Wir nehmen $\mathbf{1};\tau \xrightarrow{\omega} t'$ an. Dann ist $\omega = \tau$ und $t' = \mathbf{1};\mathbf{1}$, abgeleitet mit R_2, R_{13}. Mit der Definition von $\Rightarrow$ wissen wir, daß $\mathbf{1} \stackrel{\hat{\tau}}{\Longrightarrow} \mathbf{1}$ gilt. Weiterhin folgt aus $\mathbf{1};\mathbf{1} \sim_\beta \mathbf{1}$, daß $(\mathbf{1};\mathbf{1},\mathbf{1}) \in R$.

– Wir nehmen $\mathbf{1} \xrightarrow{\omega} t'$ an. Dann ist $\omega = \checkmark$ und $t' = \mathbf{1}$, abgeleitet mit R_1. Mit R_2, R_{13} können wir schließen, daß $\mathbf{1};\tau \xrightarrow{\tau} \mathbf{1};\mathbf{1}$ gilt. Dann wissen wir mit $\mathbf{1};\mathbf{1} \sim_\beta \mathbf{1}$, daß $(\mathbf{1};\mathbf{1},\mathbf{1}) \in R$ gilt. Weiterhin können wir mit R_1, R_{13} ableiten, daß $\mathbf{1};\mathbf{1} \xrightarrow{\checkmark} \mathbf{1};\mathbf{1}$ gilt. Daher gilt $\mathbf{1};\tau \stackrel{\widehat{\checkmark}}{\Longrightarrow} \mathbf{1};\mathbf{1}$ und $(\mathbf{1};\mathbf{1},\mathbf{1}) \in R$.

□

Proposition 3.16. $t \sim_\beta u$ impliziert $t \simeq_\beta u$, und $t \simeq_\beta u$ impliziert $t \approx_\beta u$.

Beweis für Proposition 3.16. Aus Definition 3.7 und Definition 3.12 folgt direkt, daß $t \sim_\beta u$ die Gültigkeit von $t \approx_\beta u$ impliziert. Für den Beweis, daß $t \sim_\beta u$ die Gültigkeit von $t \simeq_\beta u$ impliziert, können wir Definition 3.15 und die Definition von $\Rightarrow$ verwenden: $\forall \omega \in \Omega$: Wenn $t \xrightarrow{\omega} t'$, dann $t \stackrel{\omega}{\Longrightarrow} t'$. Für den Beweis, daß $t \simeq_\beta u$ die Gültigkeit von $t \approx_\beta u$ impliziert, können wir die Kongruenzeigenschaft von $\simeq_\beta$ verwenden: Aus $t \simeq_\beta u$ folgt $t + \mathbf{0} \simeq_\beta u + \mathbf{0}$. Weiterhin können wir mit $t + \mathbf{0} \sim_\beta t$ und $u + \mathbf{0} \sim_\beta u$ ableiten, daß $t + \mathbf{0} \approx_\beta t$ und $u + \mathbf{0} \approx_\beta u$ gilt. Aus $\mathbf{0} \not\xrightarrow{}$ und $t \simeq_\beta u$ folgt dann $t \approx_\beta u$. □

Theorem 3.17. $\simeq_\beta$ ist die größte in $\approx_\beta$ enthaltene Kongruenz für die Operatoren aus $\mathcal{P}$.

Für den Beweis des Theorems benötigen wir das folgende Lemma:

Lemma 3.21 *Wenn $\sigma =_\beta \sigma'$, dann $t\sigma \simeq_\beta t\sigma'$.*

Beweis für Lemma 3.21. Folgt unmittelbar aus Lemma 3.9 und $\sim_\beta \subset \simeq_\beta$ (Proposition 3.16). □

Beweis für Theorem 3.17.

- $spawn(t) \simeq_\beta spawn(u)$ falls $t \simeq_\beta u$.

  Sei $R = \{(spawn(t_0), spawn(u_0)) \mid t_0 \simeq_\beta u_0\}$. Der Beweis, daß R die Anforderungen aus Definition 3.15 erfüllt, entspricht dem Beweis aus Theorem 3.13. Weiterhin können wir Theorem 3.13 verwenden, um zu zeigen, daß die nach Ausführung der initialen Transitionen enstehenden Restterme schwach bisimulär sind.

- $\{\vec{x} \leftarrow \vec{E}\};t \simeq_\beta \{\vec{x} \leftarrow \vec{E}'\};u$ falls $t \simeq_\beta u$ und $\vec{E} =_\beta \vec{E}'$.

  Sei $R = \{(\{\vec{x} \leftarrow \vec{E}\};t_0, \{\vec{x} \leftarrow \vec{E}'\};u_0) \mid t_0 \simeq_\beta u_0, \vec{E} =_\beta \vec{E}'\}$. Der Beweis, daß R die Anforderungen aus Definition 3.15 erfüllt, entspricht dem Beweis aus

Theorem 3.13. Weiterhin können wir Theorem 3.13 verwenden, um zu zeigen, daß die nach Ausführung der initialen Transitionen enstehenden Restterme schwach bisimulär sind. Weiterhin können wir Lemma 3.21 anwenden, um zu zeigen, daß die Prämissen von R_6 erfüllt sind.

- $E?x;t \simeq_\beta E'?x;u$ if $t \simeq_\beta u$ and $E =_\beta E'$.

  Sei $R = \{(E?x;t_0, E'?x;u_0) \mid t_0 \simeq_\beta u_0, E =_\beta E'\}$. Der Beweis, daß R die Anforderungen aus Definition 3.15 erfüllt, entspricht dem Beweis aus Theorem 3.13. Wir können Theorem 3.13 und Lemma 3.21 verwenden, um zu zeigen, daß die nach Ausführung der initialen Transitionen enstehenden Restterme schwach bisimulär sind.

- $\mathbf{ch}\ C.\ t \simeq_\beta \mathbf{ch}\ C.\ u$ if $t \simeq_\beta u$.

  Sei $R = \{(\mathbf{ch}\ D.\ t_0, \mathbf{ch}\ D.\ u_0) \mid t_0 \simeq_\beta u_0\}$. Der Beweis, daß R die Anforderungen aus Definition 3.15 erfüllt, entspricht dem Beweis aus Theorem 3.13. Weiterhin können wir Theorem 3.13 verwenden, um zu zeigen, daß die nach Ausführung der initialen Transitionen enstehenden Restterme schwach bisimulär sind.

- $t_1;t_2 \simeq_\beta u_1;u_2$ if $t_1 \simeq_\beta u_1$ and $t_2 \simeq_\beta u_2$.

  Sei $R = \{(t_1;t_2, u_1;u_2) \mid t_1 \simeq_\beta u_1, t_2 \simeq_\beta u_2\}$. Der Beweis, daß R die Anforderungen aus Definition 3.15 erfüllt, entspricht dem Beweis aus Theorem 3.13. Weiterhin können wir Theorem 3.13 verwenden, um zu zeigen, daß die nach Ausführung der initialen Transitionen enstehenden Restterme schwach bisimulär sind.

- $t_1 + t \simeq_\beta t_2 + t$ if $t_1 \simeq_\beta t_2$.

  Sei $R = \{(u_1 + u, u_2 + u) \mid u_1 \simeq_\beta u_2\}$. Wir zeigen, daß R die Anforderungen aus Definition 3.15 erfüllt. Wegen der Symmetrie von R zeigen wir den Beweis nur für eine Richtung.

  Wir nehmen $u_1 + u \xrightarrow{\omega} u_{new}$ an und unterscheiden die folgenden Fälle:

  - $u_1 \xrightarrow{c?v} u_1'$, $\omega = c?v$ und $u_{new} = u_1'$, abgeleitet mit R_9. Aus $u_1 \simeq_\beta u_2$ folgt $\forall v', v =_\beta v', \exists u_2' : u_2 \xRightarrow{c?v'} u_2'$ und $u_1' \approx_\beta u_2'$. Dann können wir mit R_9 folgern, daß $u_2 + u \xRightarrow{c?v'} u_2'$ gilt. Mit $u_1' \approx_\beta u_2'$ sind die Anforderungen aus Definition 3.15 erfüllt.
  - $u_1 \xrightarrow{\omega} u_1'$, $\omega \neq c?v$ und $u_{new} = u_1'$, abgeleitet mit R_9. Aus $u_1 \simeq_\beta u_2$ folgt $\exists \omega', \omega =_\beta \omega', \exists u_2' : u_2 \xRightarrow{\omega'} u_2'$ und $u_1' \approx_\beta u_2'$. Dann können wir mit R_9 folgern, daß $u_2 + u \xRightarrow{\omega'} u_2'$ gilt. Mit $u_1' \approx_\beta u_2'$ sind die Anforderungen aus Definition 3.15 erfüllt.

– $u \xrightarrow{\omega} u'$ und $u_{new} = u'$, abgeleitet mit R_{10}. Ebenso können wir mit R_{10} ableiten, daß $u_2 + u \xrightarrow{\omega} u'$ und damit auch $u_2 + u \stackrel{\omega}{\Longrightarrow} u'$ gilt. Weiterhin gilt $u' \approx_\beta u'$.

□

Theorem 3.18. Die axiomatische Theorie $\mathcal{AX}_{\simeq_\beta}$ ist korrekt bezüglich $\simeq_\beta$: Für $\mathcal{AX}_{\simeq_\beta} \vdash t = u$ gilt $t \simeq_\beta u$.

Beweis für Theorem 3.18. Für jedes Axiom aus Abbildung 3.6 definieren wir eine Relation und zeigen, daß sie die Anforderungen aus Definition 3.15 erfüllt. Wie in Abbildung 3.3 gefordert, nehmen wir für jeden Transitionsschritt $t \xrightarrow{c!v,C} t'$ an, daß $C \cap fc(t) = \emptyset$ gilt.

- Axiom (3.37): $\gamma; \tau = \gamma$.

  Sei $R = \{(\gamma; \tau, \gamma)\}$. Wir zeigen, daß R die Anforderungen aus Definition 3.15 erfüllt. Hierfür unterscheiden wir die folgenden Fälle:

  – $\gamma; \tau \xrightarrow{\gamma} \mathbf{1}; \tau$, abgeleitet mit R_{12}. Aus $\gamma \xrightarrow{\gamma} \mathbf{1}$ folgt $\gamma \stackrel{\gamma}{\Longrightarrow} \mathbf{1}$. Weiterhin wissen wir mit Proposition 3.14, daß $\mathbf{1}; \tau \approx_\beta \mathbf{1}$ gilt.

  – $\gamma \xrightarrow{\gamma} \mathbf{1}$. Dann gilt $\gamma; \tau \xrightarrow{\gamma} \mathbf{1}; \tau$, abgeleitet mit R_{12}. Mit Proposition 3.14 wissen wir $\mathbf{1}; \tau \approx_\beta \mathbf{1}$. Weiterhin können wir mit R_2 und R_{13} ableiten, daß $\mathbf{1}; \tau \xrightarrow{\tau} \mathbf{1}; \mathbf{1}$ und $\mathbf{1}; \mathbf{1} \approx_\beta \mathbf{1}$ gilt. Daher gilt $\gamma; \tau \stackrel{\gamma}{\Longrightarrow} \mathbf{1}; \mathbf{1}$ und $\mathbf{1}; \mathbf{1} \approx_\beta \mathbf{1}$.

- Axiom (3.38): $spawn(\tau; t) = \tau; spawn(t)$.

  Sei $R = \{(spawn(\tau; u), \tau; spawn(u)) \mid u \in \mathcal{P}\}$. Wir zeigen, daß R die Anforderungen aus Definition 3.15 erfüllt. Hierfür unterscheiden wir die folgenden Fälle:

  – Wir nehmen $spawn(\tau; u) \xrightarrow{\omega} u'$ an und unterscheiden die folgenden Fälle:

    * $\omega = \tau$ und $u' = spawn(\mathbf{1}; u)$, abgeleitet mit R_2, R_{12}, R_8. Ebenso können wir mit R_2, R_{12} ableiten, daß $\tau; spawn(u) \xrightarrow{\tau} \mathbf{1}; spawn(u)$ gilt. Weiterhin gilt $spawn(\mathbf{1}; u) \approx_\beta \mathbf{1}; spawn(u)$.
    * $\omega = \checkmark$ und $u' = spawn(\tau; u)$, abgeleitet mit R_7. Mit R_2,R_{12} können wir schließen, daß $\tau; spawn(u) \xrightarrow{\tau} \mathbf{1}; spawn(u)$ gilt. Weiterhin wissen wir mit Proposition 3.14, daß $\mathbf{1}; spawn(u) \approx_\beta spawn(u)$ und $spawn(\tau; u) \approx_\beta spawn(u)$ gilt. Dann können wir mit R_1, R_7, R_{13} ableiten, daß $\mathbf{1}; spawn(u) \xrightarrow{\checkmark} \mathbf{1}; spawn(u)$ und daher auch $\tau; spawn(u) \stackrel{\checkmark}{\Longrightarrow} \mathbf{1}; spawn(u)$ gilt. Weiterhin wissen wir mit $\mathbf{1}; spawn(u) \approx_\beta spawn(u)$, daß $spawn(\tau; u) \approx_\beta \mathbf{1}; spawn(u)$ gilt.

  - Wir nehmen $\tau; \mathit{spawn}(u) \xrightarrow{\omega} u'$ an. Dann wissen wir mit R_2, R_{12}, daß $\omega = \tau$ und $u' = \mathbf{1}; \mathit{spawn}(u)$ gilt. Ebenso können wir mit R_2, R_{12}, R_8 ableiten, daß $\mathit{spawn}(\tau; u) \xrightarrow{\tau} \mathit{spawn}(\mathbf{1}; u)$ gilt. Weiterhin gilt $\mathit{spawn}(\mathbf{1}; u) \approx_\beta \mathbf{1}; \mathit{spawn}(u)$.

- Axiom (3.39): $t + \tau; t = \tau; t$.
  Sei $R = \{(u + \tau; u, \tau; u) \mid u \in \mathcal{P}\}$. Wir zeigen, daß R die Anforderungen aus Definition 3.15 erfüllt.

  - Wir nehmen $u+\tau; u \xrightarrow{\omega} u_{new}$ an und unterscheiden die folgenden Fälle:
    * $u \xrightarrow{\omega} u'$ and $u_{new} = u'$, abgeleitet mit R_9. Dann können wir mit R_2, R_{12} ableiten, daß $\tau; u \xrightarrow{\tau} \mathbf{1}; u$ gilt. Aus $u \sim_\beta \mathbf{1}; u$ und Proposition 3.16 folgt $u \approx_\beta \mathbf{1}; u$. Weiterhin können wir mit R_{13} ableiten, daß $\mathbf{1}; u \xrightarrow{\omega} \mathbf{1}; u'$ und $u' \approx_\beta \mathbf{1}; u'$ gilt. Daher gilt auch $\tau; u \xRightarrow{\gamma} \mathbf{1}; u'$ und $u' \approx_\beta \mathbf{1}; u'$.
    * $\tau; u \xrightarrow{\tau} \mathbf{1}; u$, $\omega = \tau$ und $u_{new} = \mathbf{1}; u$, abgeleitet mit R_2, R_{12}, R_{10}. Ebenso können wir mit R_2, R_{12} ableiten, daß $\tau; u \xrightarrow{\tau} \mathbf{1}; u$ und daher auch $\tau; u \xRightarrow{\tau} \mathbf{1}; u$ gilt. Weiterhin gilt $\mathbf{1}; u \approx_\beta \mathbf{1}; u$.
  - Wir nehmen $\tau; u \xrightarrow{\omega} u_{new}$ an. Dann gilt $\omega = \tau$ und $u_{new} = \mathbf{1}; u$, abgeleitet mit R_2, R_{12}. Dann können wir mit R_2, R_{12}, R_{10} ableiten, daß $u + \tau; u \xrightarrow{\tau} \mathbf{1}; u$ und daher auch $u + \tau; u \xRightarrow{\tau} \mathbf{1}; u$ gilt. Weiterhin gilt $\mathbf{1}; u \approx_\beta \mathbf{1}; u$.

- Axiom (3.40): $E?x; \tau; t = E?x; t$.
  Sei $R = \{(E?x; \tau; u, E?x; u) \mid u \in \mathcal{P}\}$. Wir zeigen, daß R die Anforderungen aus Definition 3.15 erfüllt. Wir nehmen $\mathit{red}[\![E]\!] = c$ an.

  - $E?x; \tau; u \xrightarrow{c?v} (\tau; u)\langle v/x\rangle$, abgeleitet mit R_5. Aus $\forall \sigma : \tau\sigma = \tau$ folgt $(\tau; u)\langle v/x\rangle = \tau; u\langle v/x\rangle$. Ebenso können wir mit R_5 ableiten, daß $E?x; u \xrightarrow{c?v} u\langle v/x\rangle$ und daher auch $E?x; u \xrightarrow{c?v} u\langle v/x\rangle$ gilt. Weiterhin wissen wir mit Proposition 3.14, daß $\tau; u\langle v/x\rangle \approx_\beta u\langle v/x\rangle$ gilt.
  - Für $E?x; u \xrightarrow{c?v} u\langle v/x\rangle$ analog.

□

Kapitel 4

Integration einer Datensprache

In diesem Kapitel beschreiben wir einen erweiterten getypten λ-Kalkül [Bar84, Bar90], den wir als Datenteilsprache in den in Kapitel 3 vorgestellen Prozeßkalkül integrieren. Die so entstandene Sprache $\mathcal{P}_\lambda$ wenden wir im Rahmen von zwei Fallstudien an. In der zweiten Fallstudie verwenden wir die axiomatische Theorie aus Kapitel 3, um zu zeigen, daß zwei verschiedene Spezifikationen der Fallstudie äquivalentes Verhalten zeigen. Wie in Kapitel 3 geben wir die Beweise für die Theoreme in einem speziellen Abschnitt 4.5 an.

4.1 Syntax und Typsystem

Zunächst definieren wir die Syntax und das Typsystem der Datenteilsprache. Hierzu konkretisieren wir die in Abschnitt 3.1 definierten Begriffe. Zuerst beschreiben wir die Menge DTYPE der Typausdrücke. Dann definieren wir die Syntax der Datensprache. Anschließend geben wir ein Typsystem an, dessen Typregeln die Typisierung der Ausdrücke der Datensprache beschreiben.

Typausdrücke. Die Menge DTYPE der Typausdrücke der Datentypen wird durch die folgende Grammatik definiert:

$$\begin{array}{lcl} ET & ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{unit} \mid T\ \mathit{channel} \mid ET \times \cdots \times ET \mid ET\ \mathit{list} \\ T & ::= & ET \mid T \times \cdots \times T \mid T\ \mathit{list} \mid T \to T \end{array}$$

Wir unterscheiden zwischen Typen, für die ein Gleichheitsbegriff existiert, und solchen, für die ein solcher Begriff nicht existiert. Hierbei bezeichnet ET die Teilmenge der Typausdrücke mit Gleichheitsbegriff. Hierzu zählen alle Typausdrücke für Typen, deren Werte nicht Funktionen sind bzw. keine Funktionen als Komponenten enthalten. T repräsentiert beliebige Typausdrücke. Als Basistypen (bzw. elementare Typen) verwenden wir *int* für ganze Zahlen, *bool* für Wahrheitswerte und *unit* als den "leeren" Typ: $\textsc{Value}^{bool} = \{false, true\}$, $\textsc{Value}^{int} = \{..., -1, 0, 1, ...\}$ und $\textsc{Value}^{unit} = \{()\}$. Werte des Types T *channel*

sind Kanäle, über die Werte des Typs T übertragen werden können. Analog bilden Werte vom Typ T *list* Listen von Werten des Typs T. $T \to T'$ bezeichnet einen Funktionstyp, dessen Elemente ein Argument des Typs T nehmen und ein Ergebnis des Typs T' liefern. Funktionstypen sind rechtsassoziativ, daher entspricht $T_1 \to T_2 \to T_3$ dem Typausdruck $T_1 \to (T_2 \to T_3)$. Der Typausdruck $T_1 \times ... \times T_n$ bezeichnet die Menge der Tupel mit n Komponenten; Werte dieses Typs werden als $v_1, ..., v_n$ dargestellt.

Syntax. Die Datenteilsprache, also die Menge EXPR der Datenausdrücke, wird durch die folgende kontextfreie Grammatik definiert:

$$
\begin{array}{rcl}
E & ::= & v \mid x \mid op_{un}\, E \mid E\, op_{bin}\, E \mid E\, E \mid \mathit{if}\ E\ \mathit{then}\ E\ \mathit{else}\ E \mid \\
 & & \text{let}\ x = E\ \text{in}\ E \mid \mathit{cons}\, E\, E \mid E, ..., E \mid (E) \\
v & ::= & \mathit{const} \mid \mathit{cons}\, v\, v \mid v, \ldots, v \mid \lambda x : T.E \\
\mathit{const} & ::= & () \mid \mathit{true} \mid \mathit{false} \mid \mathit{nil} \mid n \mid c \\
op_{un} & ::= & \mathit{not} \mid \neg \mid \mathit{head} \mid \mathit{tail} \mid Y \mid \#i \\
op_{bin} & ::= & + \mid - \mid * \mid / \mid \mathit{mod} \mid \mathit{and} \mid \mathit{or} \mid < \mid > \mid =
\end{array}
$$

Das Nichtterminal v repräsentiert die Werte unserer Sprache, d.h. die Elemente der Menge VALUE. *const* definiert die Konstanten: () ist der einzige Wert des Typs *unit*, n steht für ganze Zahlen und c für Kanalwerte. op_{un} und op_{bin} stehen für die vordefinierten unären bzw. binären Operatoren und Funktionen unserer Sprache. Hierbei handelt es sich um die aus anderen funktionalen Sprachen bekannten arithmetischen und logischen Operatoren. Binäre Operatoren schreiben wir in Infix-Notation. Mit $\neg$ bezeichnen wir die Negation ganzer Zahlen. Die Typen der Operatoren sind in Abbildung 4.1 angegeben. Wir nehmen an, daß eine Menge von Funktionen $\#i$ vordefiniert ist, mit denen auf die i-te Komponente eines Tupels zugegriffen werden kann. Listen werden mit den Konstruktoren *nil* und $\mathit{cons}\, E_1\, E_2$ gebildet, wobei durch *cons* der Wert des Ausdrucks E_1 als neues Kopfelement an die durch den Ausdruck E_2 berechnete Liste angehängt wird. Wir schreiben Listenausdrücke der Form $\mathit{cons}\ E_1\ (\mathit{cons}\ E_2 ... (\mathit{cons}\ E_n\ \mathit{nil}) ...)$ als $[E_1, ..., E_n]$. Die Funktionen *head* und *tail* berechnen das erste Element bzw. den Listenrest einer Liste. Mit $\text{let}\ x = E_1\ \text{in}\ E_2$ definieren wir einen lokalen Bezeichner x, an den der Wert des Ausdrucks E_1 gebunden wird. Der Gültigkeitsbereich von x erstreckt sich auf den Ausdruck E_2.

Funktionen schreiben wir als $\lambda x : T.E$, wobei das Argument der Funktion an den Bezeichner x vom Typ T gebunden wird und im Funktionsrumpf E verwendet werden kann. Der Ausdruck E berechnet das Ergebnis der Funktion. Die Applikation von Funktionen auf ihr Argument schreiben wir als $E_1\ E_2$. Hierbei berechnet E_1 die anzuwendende Funktion, während E_2 das Argument berechnet. Funktionsapplikationen der Form $E_1\, E_2 \ldots E_n$ werden linksassoziativ als $(...(E_1\ E_2)\ ...)E_n$ interpretiert und nach dem Prinzip des *currying* [Thi94] ausgewertet: In jedem Auswertungsschritt wird eine neue Funktion generiert, die

auf die restlichen Argumente angewendet wird. So wird der Ausdruck let $f = \lambda x : int.\lambda y : int.x + y$ in $f\ 1\ 2$ wie folgt ausgewertet:

$$\begin{array}{ll} & \text{let } f = \lambda x : int.\lambda y : int.x + y \text{ in } f\ 1\ 2 \\ \hookrightarrow & (\lambda x : int.\lambda y : int.x + y)\ 1\ 2 \\ \hookrightarrow & (\lambda y : int.1 + y)\ 2 \\ \hookrightarrow & 1 + 2 \\ \hookrightarrow & 3 \end{array}$$

Die Funktion f hat den Typ $int \rightarrow int \rightarrow int$.

In funktionalen Sprachen wie z.B. *Standard ML* wird eine rekursive Funktion zur Berechnung der Fakultät wie folgt definiert:

$$fun\ fakul\ n = if\ n = 0\ \ then\ 1\ else\ n * fakul(n - 1).$$

Die Funktion erhält einen Namen $fakul$. Dieser Name wird im Ausdruck zur Berechnung des Funktionsergebnisses verwendet, um einen rekursiven Aufruf zu beschreiben. Wird die Definition als Bindung eines Funktionswertes an den Bezeichner $fakul$ interpretiert, bedeutet dies, daß der Name $fakul$ bereits im Funktionsausdruck verwendet wird, bevor er durch die Bindung erzeugt wird.

In unserem erweiterten λ-Kalkül ist, wie im Standard-λ-Kalkül, die Verwendung eines Namens vor seiner Erzeugung nicht möglich. Daher verwenden wir zur Definition rekursiver Ausdrücke den Y-Kombinator [Bar90]. Dieser erlaubt die Definition rekursiver Funktionen ohne die Verwendung eines Namens vor seiner Erzeugung. Der Y-Kombinator realisiert Rekursion, indem bei jedem Aufruf des Kombinators die rekursive Funktion als zusätzliches Argument mit übergeben wird. Der Kombinator ist so definiert, daß er wiederum diese übergebene Funktion auf ihre jeweiligen Argumente anwendet. In Sprachen mit der Auswertungsstrategie der *lazy evaluation* [Tur85, P$^+$96], bei der Ausdrücke nur dann reduziert werden, wenn ihr Wert benötigt wird, läßt sich der Y-Kombinator als $Y\ f = f\ (Y\ f)$ definieren. In Sprachen mit der Strategie der *eager evaluation* [MTH90] kann der Kombinator als $Y\ f\ x = (f\ (Y\ f))\ x$ implementiert werden.

Eine rekursive Version der Fakultätsfunktion unter Verwendung des Y-Kombinators läßt sich wie folgt definieren:

$$\begin{array}{l} \text{let } fakul_0 = \\ \quad \lambda f : int \rightarrow int. \\ \quad \lambda n : int. \\ \qquad if\ n = 0\ then\ 1\ else\ n * f(n - 1) \\ \text{in} \\ \quad \text{let } fakul = Y\ fakul_0 \\ \quad \text{in } fakul\ 6 \end{array}$$

not	:	$bool \to bool$
$\neg$	:	$int \to int$
and, or	:	$(bool \times bool) \to bool$
$+, -, *, /, mod$	:	$(int \times int) \to int$
$<, >$	:	$(int \times int) \to bool$
$=$	:	$(ET \times ET) \to bool$

Abbildung 4.1: Typen der Operatoren.

Es wird eine Hilfsfunktion an den Bezeichner $fakul_0$ gebunden, die als erstes Argument eine Funktion erwartet, mit der die Rekursion fortgeführt werden soll. Ist das zweite Argument ungleich 0, wird der rekursive Aufruf durchgeführt. Die Funktion $fakul$ entsteht nun aus der Anwendung des Y-Kombinators auf $fakul_0$. Der Kombinator sorgt nun dafür, daß bei jedem Aufruf von $fakul_0$ eine Kopie von $Y\ fakul_0$ als erstes Argument übergeben wird, so daß die Möglichkeit von weiteren rekursiven Aufrufen von $fakul_0$ gegeben ist. Die Auswertung des obigen Ausdrucks beginnt mit

$$fakul\ 6 \hookrightarrow (Y\ fakul_0)\ 6 \hookrightarrow (fakul_0\ (Y\ fakul_0))\ 6$$
$$\hookrightarrow if\ \ 6 = 0\ then\ \ 1\ else\ \ 6 * (Y\ fakul_0)\ (6-1) \hookrightarrow ...$$

Der Y-Kombinator muß als zusätzlicher Operator in die Sprache eingeführt werden, da die oben angegebenen Definitionen des Kombinators in Form von Funktionen selbst rekursive Funktionen sind. Daher ist der Kombinator mit den Konstrukten unserer Sprache nicht darstellbar.

Zur Vereinfachung der Darstellung von Ausdrücken führen wir Prioritäten für die Operationen der Datensprache ein. Wir legen fest, daß die Funktionsapplikation eine höhere Priorität als die anderen Operatoren besitzt. Dies beinhaltet auch die Verwendung der vordefinierten Funktionen wie Y und $cons$. Weiterhin besitzen die unären und binären Operatoren eine höhere Priorität als die lokale Definition von Bezeichnern mit let, die Fallunterscheidung mit if und die Funktionsabstraktion.

Typsystem. Wir definieren das Typsystem für die Datenteilsprache als Erweiterung der Typregeln aus Abbildung 3.2. Die Typregeln für die Datenteilsprache sind in Abbildung 4.2 angegeben. Die Operatoren aus Abbildung 4.1 behandeln wir als Funktion $op\,E$ für unäre Operatoren bzw. als $op\,(E_1, E_2)$ für binäre Operatoren. Regel T_{15} ist die Abschwächungsregel für Ausdrücke (vergleiche T_{13} in Abbildung 3.2). Die Regeln T_{16} bis T_{20} definieren die Typen der Konstanten. Regel T_{21} typisiert bedingte Ausdrücke. Die Regeln T_{22} und T_{23} definieren die

$$\frac{}{x : T \vdash x : T}\,\mathrm{T}_{14} \qquad \frac{\Delta' \vdash E : T \quad \Delta' \subseteq \Delta}{\Delta \vdash E : T}\,\mathrm{T}_{15} \qquad \frac{}{\vdash n : \mathit{int}}\,\mathrm{T}_{16}$$

$$\frac{c \in \textsc{Chan}^T}{\vdash c : T\ \mathit{channel}}\,\mathrm{T}_{17} \qquad \frac{}{\vdash \mathit{false} : \mathit{bool}}\,\mathrm{T}_{18} \qquad \frac{}{\vdash \mathit{true} : \mathit{bool}}\,\mathrm{T}_{19}$$

$$\frac{}{\vdash () : \mathit{unit}}\,\mathrm{T}_{20} \qquad \frac{\Delta \vdash E_1 : \mathit{bool} \quad \Delta \vdash E_2 : T \quad \Delta \vdash E_3 : T}{\Delta \vdash \mathit{if}\ E_1\ \mathit{then}\ E_2\ \mathit{else}\ E_3 : T}\,\mathrm{T}_{21}$$

$$\frac{\Delta \vdash E_1 : T_1 \quad \cdots \quad \Delta \vdash E_n : T_n}{\Delta \vdash (E_1, \ldots, E_n) : T_1 \times \cdots \times T_n}\,\mathrm{T}_{22} \qquad \frac{\Delta \vdash E : T_1 \times \cdots \times T_n \quad 1 \leq i \leq n}{\Delta \vdash \#i\ E : T_i}\,\mathrm{T}_{23}$$

$$\frac{}{\vdash \mathit{nil} : T\ \mathit{list}}\,\mathrm{T}_{24} \qquad \frac{\Delta \vdash E_1 : T \quad \Delta \vdash E_2 : T\ \mathit{list}}{\Delta \vdash \mathit{cons}\ E_1\ E_2 : T\ \mathit{list}}\,\mathrm{T}_{25}$$

$$\frac{\Delta \vdash E : T\ \mathit{list}}{\Delta \vdash \mathit{head}\ E : T}\,\mathrm{T}_{26} \qquad \frac{\Delta \vdash E : T\ \mathit{list}}{\Delta \vdash \mathit{tail}\ E : T\ \mathit{list}}\,\mathrm{T}_{27}$$

$$\frac{\Delta, x : T' \vdash E : T}{\Delta \vdash (\lambda x.E) : T' \to T}\,\mathrm{T}_{28} \qquad \frac{\Delta \vdash E : T' \to T \quad \Delta \vdash E' : T'}{\Delta \vdash E\ E' : T}\,\mathrm{T}_{29}$$

$$\frac{\Delta \vdash E_1 : T' \quad \Delta, x : T' \vdash E_2 : T}{\Delta \vdash \mathrm{let}\ \ x = E_1\ \ \mathrm{in}\ \ E_2 : T}\,\mathrm{T}_{30} \qquad \frac{\Delta \vdash E : (T \to T') \to T \to T'}{\Delta \vdash Y\ E : T \to T'}\,\mathrm{T}_{31}$$

Abbildung 4.2: Typregeln der funktionalen Datensprache.

Typisierung von Tupeln bzw. der Zugriffsfunktionen auf einzelne Elemente eines Tupels. Die Regel T_{24} definiert, daß *nil* als Konstante jedes beliebigen Listentyps interpretiert werden kann. Mit den Regeln T_{25} bis T_{27} werden die Funktionen auf Listen typisiert. T_{28} beschreibt die Typisierung von Funktionen, T_{29} von Funktionsapplikationen. Mit T_{30} werden *let*-Ausdrücke getypt. Die Regel T_{31} definiert den Typ des Y-Kombinators. Sie legt fest, daß der Ausdruck E, auf den Y angewendet wird, eine Funktion sein muß, die als erstes Argument eine Funktion des Typs $T \to T'$ sowie ein Argument des Typs T erwartet und ein Ergebnis des Typs T' zurückliefert. Der Ausdruck $Y\,E$ hat dann den Typ $T \to T'$. So hat im Beispiel der Fakultätsfunktion die Hilfsfunktion $fakul_0$ den Typ $(\mathit{int} \to \mathit{int}) \to \mathit{int} \to \mathit{int}$ und die Funktion $fakul$ den Typ $\mathit{int} \to \mathit{int}$, was dem erwarteten Typ der Fakultätsfunktion entspricht.

4.2 Reduktionssemantik

Nun definieren wir die Reduktionssemantik für Ausdrücke. Wir geben Reduktionsregeln für geschlossene Ausdrücke an, die die Auswertung der einzelnen Operatoren der Datenteilsprache beschreiben.

Definition 4.1 *Sei* CExpr $\subseteq$ Expr *die Menge der geschlossenen Ausdrücke, d.h. der Ausdrücke ohne freie Bezeichner. Die* Reduktionsrelation $\hookrightarrow \subseteq$ CExpr $\times$ CExpr *wird durch die Reduktionsregeln in Abbildung 4.3 definiert. Die reflexive und transitive Hülle von* $\hookrightarrow$ *bezeichnen wir mit* $\hookrightarrow^*$. *Ein Wert* v *ist die* Reduktionssemantik *eines Ausdrucks* E, *falls* $E \hookrightarrow^* v$. *Wir schreiben dann* $[\![E]\!] = v$.

Die Regeln R_{17} bis R_{19} beschreiben die Auswertung einer Fallunterscheidung. Zuerst wird die Bedingung zu einem Wert reduziert. Anschließend wird die Reduktion in Abhängigkeit der Bedingung mit einem der beiden Ausdrücke E_1 oder E_2 fortgesetzt. Regel R_{20} beschreibt, daß die Teilausdrücke zur Berechnung von Elementen eines Tupels von links nach rechts ausgewertet werden. Regel R_{21} beschreibt das Lesen eines Tupel-Elements mit der Indexfunktion $\#i$. Die Regeln R_{22} bis R_{27} definieren die Reduktion der Listenfunktionen. Gemäß der *eager evaluation*-Auswertungsstrategie werden jeweils zuerst die Argumente ausgewertet, bevor die Funktionsapplikation durchgeführt wird. Die Regeln R_{28}, R_{29} und R_{30} definieren die Funktionsapplikation. Zunächst wird der Ausdruck berechnet, der die anzuwendende Funktion liefert. Anschließend wird das Argument der Funktionsapplikation berechnet. Nachdem die Ausdrücke berechnet wurden, wird mit R_{30} die eigentliche Applikation durchgeführt, die der β-Reduktion des λ-Kalküls entspricht. Die Regeln R_{31} und R_{32} beschreiben die Reduktion von Ausdrücken mit lokalen Definitionen. Durch die Regeln R_{33} bis R_{35} wird die Auswertung des Y-Kombinators definiert. Mit Hilfe der ersten beiden Regeln werden die Argumente des Kombinators reduziert. Liegen die Argumente als Werte vor, wird der Kombinator ausgewertet. Aufgrund der für unsere Datensprache gewählte Auswertungsstrategie der *eager evaluation* entspricht die Reduktion des Kombinators der auf Seite 93 angegebenen Definition in funktionalen Sprachen mit dieser Strategie.

Analog zum *subject reduction*-Theorem 3.5 für Prozeßterme besagt das folgende Theorem, daß die Reduktion eines wohlgetypten Ausdrucks ebenfalls ein wohlgetypter Ausdruck ist. Daher ist die Menge der wohlgetypten Ausdrücke unter Reduktion abgeschlossen.

Theorem 4.2 *Wenn* $\vdash E : T$ *und* $E \hookrightarrow E'$, *dann* $\vdash E' : T$.

Wir benötigen noch die Definition der Funktion fc zur Berechnung der freien Kanalwerte eines Ausdrucks sowie die Äquivalenzrelation $=_\beta$.

$$\frac{E \hookrightarrow E'}{\mathit{if}\ E\ \mathit{then}\ E_1\ \mathit{else}\ E_2 \hookrightarrow \mathit{if}\ E'\ \mathit{then}\ E_1\ \mathit{else}\ E_2}\,\mathrm{R}_{17}$$

$$\frac{}{\mathit{if}\ \mathit{true}\ \mathit{then}\ E_1\ \mathit{else}\ E_2 \hookrightarrow E_1}\,\mathrm{R}_{18} \qquad \frac{}{\mathit{if}\ \mathit{false}\ \mathit{then}\ E_1\ \mathit{else}\ E_2 \hookrightarrow E_2}\,\mathrm{R}_{19}$$

$$\frac{E_i \hookrightarrow E_i' \quad (1 \leq i \leq n)}{(v_1, \ldots, v_{i-1}, E_i, \ldots, E_n) \hookrightarrow (v_1, \ldots, v_{i-1}, E_i', \ldots, E_n)}\,\mathrm{R}_{20}$$

$$\frac{1 \leq i \leq n}{\#i\ (v_1, \ldots, v_n) \hookrightarrow v_i}\,\mathrm{R}_{21}$$

$$\frac{E_1 \hookrightarrow E_1'}{\mathit{cons}\ E_1\ E_2 \hookrightarrow \mathit{cons}\ E_1'\ E_2}\,\mathrm{R}_{22} \qquad \frac{E \hookrightarrow E'}{\mathit{cons}\ v\ E \hookrightarrow \mathit{cons}\ v\ E'}\,\mathrm{R}_{23}$$

$$\frac{E \hookrightarrow E'}{\mathit{head}\ E \hookrightarrow \mathit{head}\ E'}\,\mathrm{R}_{24} \qquad \frac{}{\mathit{head}\ (\mathit{cons}\ v_1\ v_2) \hookrightarrow v_1}\,\mathrm{R}_{25}$$

$$\frac{E \hookrightarrow E'}{\mathit{tail}\ E \hookrightarrow \mathit{tail}\ E'}\,\mathrm{R}_{26} \qquad \frac{}{\mathit{tail}\ (\mathit{cons}\ v_1\ v_2) \hookrightarrow v_2}\,\mathrm{R}_{27}$$

$$\frac{E \hookrightarrow E'}{E\ F \hookrightarrow E'\ F}\,\mathrm{R}_{28} \qquad \frac{F \hookrightarrow F'}{v\ F \hookrightarrow v\ F'}\,\mathrm{R}_{29} \qquad \frac{}{(\lambda x.E)\ v \hookrightarrow E\langle v/x\rangle}\,\mathrm{R}_{30}$$

$$\frac{E \hookrightarrow E'}{\text{let}\ \ x = E\ \ \text{in}\ \ F \hookrightarrow \text{let}\ \ x = E'\ \ \text{in}\ \ F}\,\mathrm{R}_{31} \qquad \frac{}{\text{let}\ \ x = v\ \ \text{in}\ \ E \hookrightarrow E\langle v/x\rangle}\,\mathrm{R}_{32}$$

$$\frac{E_1 \hookrightarrow E_1'}{Y\ E_1\ E_2 \hookrightarrow Y\ E_1'\ E_2}\,\mathrm{R}_{33} \qquad \frac{E_2 \hookrightarrow E_2'}{Y\ v\ E_2 \hookrightarrow Y\ v\ E_2'}\,\mathrm{R}_{34}$$

$$\frac{}{Y\ v_1\ v_2 \hookrightarrow (v_1(Y\ v_1))\ v_2}\,\mathrm{R}_{35}$$

Abbildung 4.3: Reduktionssemantik für funktionale Teilsprache.

Definition 4.3 *Wir definieren die partielle Funktion* $fc : \text{EXPR} \rightharpoonup 2^{\text{CHAN}}$ *zur Berechnung der Menge der freien Kanalwerte eines Ausdrucks wie folgt:*

- $fc(c) = \{c\}$
- $fc(const) = \emptyset$ *für* $const \notin \text{CHAN}$
- $fc(cons\ v_1\ v_2) = fc(v_1) \cup fc(v_2)$
- $fc(v_1, ..., v_n) = \bigcup_{1 \leq i \leq n} fc(v_i)$
- $fc(\lambda x.E) = \bigcup_v fc([\![E\ v]\!])$

Die Funktion ist nur für Werte definiert, da in der operationellen Semantik in Abbildung 3.3 und den Definitionen der Äquivalenzrelationen nur die freien Kanalwerte von Transitionsbeschriftungen betrachtet werden und das Auftreten von nicht-reduzierten Ausdrücken in Transitionsbeschriftungen nicht möglich ist. In den Gleichungen der axiomatischen Semantik in Abbildung 3.4 und Abbildung 3.5 müssen die Datenwerte vor der Anwendung einer Gleichung reduziert werden.

Die freien Kanäle eines Funktionswertes errechnen sich aus der Menge aller freien Kanäle, die bei der Anwendung der Funktion auf beliebige Werte in den Resultaten auftreten. Diese Definition der freien Kanäle unterstützt die in Abschnitt 3.1 geforderte Eigenschaft, daß aus $E =_\beta E'$ die Gültigkeit von $fc(E) = fc(E')$ folgt (siehe unten).

Nun definieren wir die Äquivalenzrelation $=_\beta$ auf Ausdrücken.

Definition 4.4 *Sei* $=_\beta \subseteq \text{CEXPR} \times \text{CEXPR}$ *eine Relation.*

- *Sind* E, E' *keine Lambda-Abstraktionen, gilt* $E =_\beta E'$ *falls* $\exists v : [\![E]\!] = v = [\![E']\!]$.
- *Sind* E, E' *Lambda-Abstraktionen, gilt* $E =_\beta E'$ *falls* $\forall v' \exists v : [\![E\ v']\!] = v = [\![E'\ v']\!]$.

Zwei offene Ausdrücke E, E' *sind* β*-äquivalent, falls* $\forall \sigma : E\sigma =_\beta E'\sigma$.

Wir unterscheiden zwei Fälle: Handelt es sich bei den zu untersuchenden Ausdrücken nicht um Funktionswerte, sind die Ausdrücke äquivalent, wenn sie sich auf denselben Wert reduzieren lassen. Handelt es sich bei den Ausdrücken um Funktionswerte, sind sie äquivalent, wenn sie für Applikationen auf beliebige Werte jeweils identische Resultate berechnen.

Die Definition von fc gewährleistet, daß aus $E =_\beta E'$ sofort $fc(E) = fc(E')$ folgt und somit diese Anforderung an die Datensprache aus Abschnitt 3.1 erfüllt wird.

4.3 Beispiele

In diesem Abschnitt wenden wir den Kalkül zur Spezifikation zweier Beispiele an. Das erste Beispiel ist die Spezifikation einer FIFO-Speicherzelle. Im zweiten Beispiel untersuchen wir eine Fallstudie, in der ein *Remote-Procedure-Call*-Speicher spezifiziert werden soll. Wir zeigen durch Anwendung der axiomatischen Theorie aus Kapitel 3, daß zwei verschiedene Spezifikationen dieser Fallstudie äquivalentes Verhalten zeigen.

4.3.1 FIFO-Speicherzelle

In diesem Abschnitt spezifizieren wir eine FIFO-Speicherzelle zur Speicherung ganzer Zahlen. Die Zelle interagiert mit ihrer Umgebung über zwei Kanäle: Auf dem Kanal *put* kann die Umgebung Werte an die Speicherzelle senden, auf *get* kann das erste Element aus der Schlange gelesen werden. Ist die Zelle leer, soll eine Kommunikation über *get* nicht möglich sein.

Ein Spezifikation der Speicherzelle in unserer um Datenoperationen erweiterten Sprache $\mathcal{P}_\lambda$ ist in Abbildung 4.4 angegeben. Die Prozeßdefinition *FIFO_Queue* besitzt drei Parameter. Die Liste *queue* enthält den aktuellen Inhalt der Speicherzelle. Die Kanäle *get* und *put* können zur Interaktion mit dem Prozeß verwendet werden. Zu Beginn des Prozeßrumpfs wird eine Funktion zur Listenkonkatenation definiert und an den Bezeichner *append* gebunden. Zur Realisierung der Rekursion wird eine Hilfsfunktion $append_0$ definiert, die neben den beiden zu verkettenden Listen eine Funktion als Argument erhält, die für den rekursiven Aufruf verwendet wird. Der Aufruf des Y-Kombinators mit $append_0$ als Argument realisiert dann die Funktion *append*.

Nach der Definition der Funktion *append* wird der Inhalt der Speicherzelle überprüft. Ist *queue* leer, ist nur das Senden von Werten über *put* möglich. Nachdem ein Wert auf *put* gelesen wurde, ruft sich der Prozeß rekursiv mit einer Liste auf, die den gelesenen Wert enthält. Enthält die Liste mindestens ein Element, ist Kommunikation auf beiden Kanälen *get* und *put* möglich. Wird ein Wert auf *put* gelesen, wird er an die Liste mit der Funktion *append* angehängt. Wird ein Wert auf *get* angefordert, wird das auf *get* gesendete Element aus der Liste entfernt. In beiden Fällen ruft sich der Prozeß rekursiv mit dem neuen Zustand des Speichers auf.

Als Beispiel für eine Interaktion mit der Speicherzelle betrachten wir den folgenden Prozeßterm:

$$\mathbf{ch}\ get, put.\ spawn(FIFO\_Queue(nil, get, put)); put!1; put!2; get?x; get?y; \mathbf{1}$$

In diesem Prozeß wird zuerst eine Instanz der Speicherzelle erzeugt. Dann werden zwei Werte an die Zelle gesendet und anschließend wieder gelesen. The Restriktion der Kanäle *get* und *put* erzwingt eine Kommunikation mit der Speicherzelle.

```
FIFO_Queue (queue : int list, get : int channel, put : int channel) ↦
  {append ←
      let append_0 =
        λf : (int list → int list → int list). λl_1 : int list.λl_2 : int list.
          if l_1 = nil then l_2 else cons (head l_1) (f (tail l_1) l_2)
      in Y append_0};
  ( [queue = nil];
        put?value; FIFO_Queue([value], get, put)
  + [not (queue = nil)];
    (       put?value; FIFO_Queue(append queue [value], get, put)
          + get!head queue; FIFO_Queue(tail queue, get, put))
  )
```

Abbildung 4.4: Spezifikation einer FIFO-Speicherzelle.

Weiterhin wird verhindert, daß andere Prozesse mit der Speicherzelle interagieren.

4.3.2 RPC-Speicher

Als zweites Beispiel spezifizieren wir eine Fallstudie, die als Basis für einen Workshop über formale Spezifikationstechniken verwendet wurde [BMS96]. In dieser Fallstudie soll ein System spezifiziert werden, das aus zwei Komponenten besteht. Die erste Komponente ist ein Speicher für ganze Zahlen und besteht aus einer Menge von Speicherzellen, die individuell für Lese- und Schreibzugriffe adressiert werden können. Die zweite Komponente ist eine *Remote-Procedure-Call*-Komponente, die die Anfragen für Speicherzugriffe entgegennimmt, an die Speicherkomponente delegiert und anschließend die von dort erhaltenen Resultate an den Aufrufer weiterleitet.

In Abbildung 4.5 ist die Struktur des Systems dargestellt. Der Benutzer sendet die Anfragen für Speicherzugriffe auf dem Kanal rc an die RPC-Komponente. Diese dekodiert die Anfragen und delegiert sie an die Speicherkomponente, indem Leseanfragen auf dem Kanal *read* und Schreibeanfragen auf *write* gesendet werden. Der Speicher besitzt eine Server-Teilkomponente, die mit der adressierten Speicherzelle kommuniziert und anschließend die von dort erhaltene Antwort über die RPC-Komponente an den Benutzer sendet.

In Abbildung 4.6 ist die Spezifikation der Speicherkomponente angegeben. Der Prozeß *MemCell* beschreibt das Verhalten einer einzelnen Speicherzelle. Die drei Parameter des Prozesses beinhalten den aktuellen Wert der Speicherzelle sowie die beiden Kanäle für Lese- bzw. Schreibzugriffe. Der Prozeß *CreateCells* dient zum Aufbau der Speicherzelle. Er erzeugt *count* Instanzen von *MemCell*,

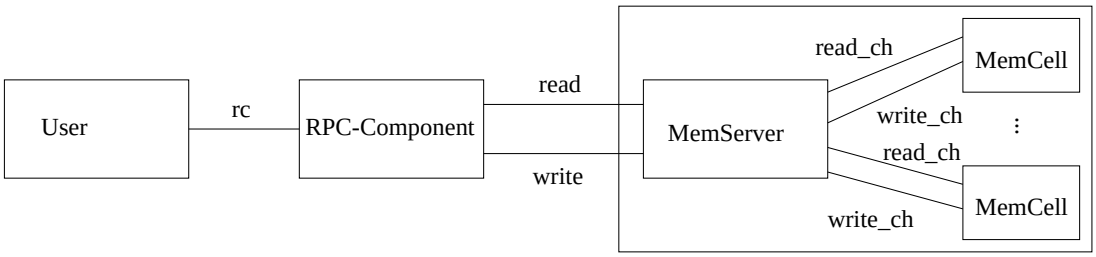


Abbildung 4.5: Struktur der RPC-Fallstudie.

wobei für jede Instanz zwei individuelle Zugriffskanäle generiert werden. Diese Kanäle werden in zwei Listen gesammelt und nach Erzeugung der Instanzen an den Aufrufer des Prozesses über den Kanal *exit* gesendet. Die Prozesse *MemRead* und *MemWrite* sind Hilfsprozesse, die dazu dienen, mehrere gleichzeitige Anfragen an die Speicherzelle parallel bearbeiten zu können. Instanzen dieser Prozesse werden beim Eintreffen einer entsprechenden Lese- oder Schreibanfrage vom Server erzeugt, so daß der eigentliche Server-Prozeß zur Annahme weiterer Anfragen bereit ist.

Der Server wird durch den Prozeß *MemServer* realisiert. Er besitzt als Parameter die beiden Listen, die die Kanäle für die Lese- und Schreibzugriffe auf die einzelnen Speicherzellen beinhalten. Über die Kanäle *read* und *write* werden die Aufträge an den Server gesendet. Zur Berechnung der Kanäle zu einer Speicherzelle wird die Funktion *nth* definiert, die das n-te Element einer Liste von Kanalwerten liefert (der Listenkopf erhält den Index 0). Anfragen auf *read* und *write* müssen Tupel der Form (adr, v, c) sein. *adr* ist die Adresse der anzusprechenden Speicherzelle, v spezifiziert den in die Zelle zu schreibenden Wert (v wird bei Leseanfragen ignoriert), und der Kanal c dient zur Übertragung des Ergebnisses an den Aufrufer. Durch die Verwendung eines speziellen Kanals für die Antwort wird die Adressierung des Auftraggebers wesentlich vereinfacht. Nachdem eine Anfrage auf *read* oder *write* empfangen wurde, wird das entsprechende Tupel zerlegt. Dann wird ein *MemRead* bzw. *MemWrite*-Prozeß erzeugt, der die Interaktion mit der adressierten Speicherzelle realisiert.

Der Prozeß *Memory* initialisiert die Speicherkomponente. Zuerst erzeugt er die einzelnen Speicherzellen und die zugehörigen Zugriffskanäle durch einen Aufruf von *CreateCells*. Danach wird eine Instanz von *MemServer* erzeugt, die die von *CreateCells* generierten Kanallisten als Parameter erhält.

Die RPC-Komponente erhält die Anfragen auf dem Kanal *rc*. Anfragen bestehen aus Tupeln der Form $(proc\_ch, adr, v, c)$. *proc_ch* ist entweder der Kanal *read* oder der Kanal *write*; die Elemente *adr*, v und c haben dieselbe Funktion wie zuvor beschrieben. Die Anfragen werden von dem Prozeß *RPC_Server* entgegengenommen, der dann eine Instanz von *RPC_Client* erzeugt, der diese Anfrage bearbeitet. Dies dient, ebenso wie das Erzeugen von *MemRead*- und *MemWrite*-

$$
\begin{array}{l}
\mathit{MemCell}\ (\mathit{value} : \mathit{int}, \mathit{readchan} : \mathit{int\ channel}, \mathit{writechan} : \mathit{int\ channel}) \mapsto \\
\quad \mathit{readchan}!\mathit{value}; \mathit{MemCell}(\mathit{value}, \mathit{readchan}, \mathit{writechan}) \\
\quad + \mathit{writechan}?\mathit{newval}; \mathit{MemCell}(\mathit{newval}, \mathit{readchan}, \mathit{writechan})
\end{array}
$$

$$
\begin{array}{l}
\mathit{CreateCells}\ (\mathit{count} : \mathit{int}, \mathit{readchans} : \mathit{int\ channel\ list}, \\
\qquad\qquad \mathit{writechans} : \mathit{int\ channel\ list}, \\
\qquad\qquad \mathit{exit} : (\mathit{int\ channel\ list} \times \mathit{int\ channel\ list})\ \mathit{channel}) \\
\quad [\mathit{count} > 0]; \\
\quad\ (\mathbf{ch}\ \mathit{rc}, \mathit{wc}.\ \mathit{spawn}(\mathit{MemCell}(0, \mathit{rc}, \mathit{wc})); \\
\qquad \mathit{CreateCells}(\mathit{count} - 1, \mathit{cons}\ \mathit{rc}\ \mathit{readchans}, \mathit{cons}\ \mathit{wc}\ \mathit{writechans}, \mathit{exit})) \\
\quad + [\mathit{count} = 0]; \mathit{exit}\ !\ (\mathit{readchans}, \mathit{writechans})
\end{array}
$$

$$
\begin{array}{l}
\mathit{MemRead}\ (\mathit{read\_ch} : \mathit{int\ channel}, v : \mathit{int}, \mathit{return\_ch} : \mathit{int\ channel}) \mapsto \\
\quad \mathit{read\_ch}?x; \mathit{return\_ch}\ !\ x
\end{array}
$$

$$
\begin{array}{l}
\mathit{MemWrite}\ (\mathit{write\_ch} : \mathit{int\ channel}, \mathit{value} : \mathit{int}, \mathit{return\_ch} : \mathit{int\ channel}) \mapsto \\
\quad \mathit{writech}\ !\ \mathit{value}; \mathit{returnch}\ !\ 1
\end{array}
$$

$$
\begin{array}{l}
\mathit{MemServer}\ (\mathit{readchans} : \mathit{int\ channel\ list}, \mathit{writechans} : \mathit{int\ channel\ list} \\
\qquad \mathit{read} : (\mathit{int} \times \mathit{int} \times \mathit{int\ channel})\ \mathit{channel}, \\
\qquad \mathit{write} : (\mathit{int} \times \mathit{int} \times \mathit{int\ channel})\ \mathit{channel}) \mapsto \\
\quad \{\mathit{nth} \leftarrow \\
\qquad \mathit{let}\ \mathit{nth}_0 = \lambda f : \mathit{int} \to \mathit{int\ channel\ list} \to \mathit{int\ channel}. \\
\qquad\qquad\qquad \lambda n : \mathit{int}.\lambda l : \mathit{int\ channel\ list}. \\
\qquad\qquad\qquad \mathit{if}\ \ n = 0\ \mathit{then}\ \ \mathit{head}\ l\ \mathit{else}\ \ f\ (n-1)\ (\mathit{tail}\ l) \\
\qquad \mathit{in}\ Y\ \mathit{nth}_0\}. \\
\quad (\quad \mathit{read}?\mathit{req}; \\
\qquad \{\mathit{read\_ch} \leftarrow \mathit{nth}\ (\#1\ \mathit{req})\ \mathit{readchans}, \mathit{return\_ch} \leftarrow \#3\ \mathit{req}\}. \\
\qquad \mathit{spawn}(\mathit{MemRead}(\mathit{read\_ch}, 0, \mathit{return\_ch})) \\
\quad +\ \ \mathit{write}?\mathit{req}; \\
\qquad \{\mathit{write\_ch} \leftarrow \mathit{nth}\ (\#1\ \mathit{req})\ \mathit{writechans}, v \leftarrow \#2\ \mathit{req}, \mathit{return\_ch} \leftarrow \#3\ \mathit{req}\}. \\
\qquad \mathit{spawn}(\mathit{MemWrite}(\mathit{write\_ch}, v, \mathit{return\_ch}))); \\
\quad \mathit{MemServer}(\mathit{readchans}, \mathit{writechans}, \mathit{read}, \mathit{write})
\end{array}
$$

$$
\begin{array}{l}
\mathit{Memory}\ (n : \mathit{int}, \mathit{read} : \mathit{int} \times \mathit{int} \times \mathit{int\ channel}, \mathit{write} : \mathit{int} \times \mathit{int} \times \mathit{int\ channel}) \mapsto \\
\quad \mathbf{ch}\ \mathit{lists\_ch}. \\
\qquad \mathit{spawn}(\mathit{CreateCells}(n, \mathit{nil}, \mathit{nil}, \mathit{lists\_ch})); \\
\qquad \mathit{lists\_ch}?\mathit{lists}; \\
\qquad\quad \mathit{spawn}(\mathit{MemServer}(\#1\ \mathit{lists}, \#2\ \mathit{lists}, \mathit{read}, \mathit{write}))
\end{array}
$$

Abbildung 4.6: Spezifikation der Speicherkomponente.

RPC_Server
$(remote\_call :$
$((int \times int \times int\ channel)\ channel \times int \times int \times int\ channel)\ channel) \mapsto$
$remote\_call?req; spawn(RPC\_Client(req)); RPC\_Server(remote\_call)$

$RPC\_Client\ (req : (int \times int \times int\ channel)\ channel \times int \times int \times int\ channel) \mapsto$
$\{proc\_ch \leftarrow \#1\ req, adr \leftarrow \#2\ req, value \leftarrow \#3\ req, return\_ch \leftarrow \#4\ req\}.$
ch $local\_ch.$
$proc\_ch\,!\,(adr, value, local\_ch); local\_ch?x; return\_ch\,!\,x$

Abbildung 4.7: Spezifikation der RPC-Komponente

Instanzen, zur Ermöglichung der parallelen Bearbeitung mehrere Anfragen. Der *RPC_Client* zerlegt die Anfrage durch einen Deklarationsoperator und sendet anschließend die Anfrage über *read* bzw. *write* an die Speicherkomponente. Für die Rückgabe des Ergebnisses von der Speicherkomponente wird ein lokaler Kanal *local_ch* erzeugt. Nachdem das Ergebnis auf *local_ch* empfangen wurde, wird es an den Aufrufer über den Kanal gesendet, der in der Anfrage an die RPC-Komponente enthalten war.

Als Beispiel für die Anwendung der axiomatischen Theorie $\mathcal{AX}_{\simeq_\beta}$ aus Abbildung 3.4, Abbildung 3.5 und Abbildung 3.6 zeigen wir nun, daß zwei Systeme äquivalentes Verhalten aufweisen. Hierzu definieren wir ein zweites Speichersystem ohne RPC-Komponente, in dem der Benutzer direkt mit der Speicherkomponente interagiert. Wir zeigen, daß das Originalsystem und das System ohne RPC-Komponente bzgl. der schwachen Kongruenzrelation äquivalentes Verhalten besitzen.

Um für beide Systeme eine einheitliche Benutzerschnittstelle zu schaffen, definieren wir einen Hilfsprozeß *RPC_Simulate*, der das System ohne RPC-Komponente ergänzt. Dieser Prozeß dient nur dazu, die Anfragen auf dem RPC-Kanal analog zum Prozeß *RPC_Client* zu zerlegen; er steuert aber im Gegensatz zu *RPC_Client* nicht die Rückgabe des Ergebnisses. Daher werden die Ergebnisse der Speicherkomponente direkt an den Benutzer gesendet.

$RPC\_Simulate\ (remote\_call :$
$((int \times int \times int\ channel)\ channel \times int \times int \times int\ channel)\ channel) \mapsto$
$remote\_call?req.$
$\{proc\_ch \leftarrow \#1\ req, adr \leftarrow \#2\ req, value \leftarrow \#3\ req, return\_ch \leftarrow \#4\ req\}.$
$spawn(proc\_ch\,!\,(adr, value, return\_ch)); RPC\_Simulate(remote\_call)$

Wir nehmen an, daß die Speicherkomponente korrekt arbeitet. Das folgende Theorem zeigt, daß unter dieser Annahme das System mit der RPC-Komponente das

System ohne RPC-Komponente simulieren kann.

Theorem 4.5 *Die Systeme*

$$\mathbf{ch}\ rc.\ spawn(RPC\_Simulate(rc)); \mathbf{ch}\ result.\ rc!(write, adr, v, result); result?x; \mathbf{1}$$

und

$$\mathbf{ch}\ rc.\ spawn(RPC\_Server(rc)); \mathbf{ch}\ result.\ rc!(write, adr, v, result); result?x; \mathbf{1}$$

zeigen bzgl. der schwachen Kongruenzrelation äquivalentes Verhalten.

4.4 Diskussion

In diesem Kapitel haben wir als Beispiel für eine funktionale Datensprache einen erweiterten, getypten λ-Kalkül vorgestellt. Für die Datensprache wurde ein Typsystem definiert. Weiterhin wurde die Auswertung von Ausdrücken durch eine Reduktionssemantik definiert. Die Datensprache wurde in den in Kapitel 3 vorgestellten Kalkül integriert; den so entstandenen Kalkül nennen wir $\mathcal{P}_\lambda$. Wir haben zwei Fallstudien mit $\mathcal{P}_\lambda$ spezifiziert. Im Rahmen der zweiten Fallstudie wurde ein Beispiel für die Anwendung der axiomatischen Semantik aus Abbildung 3.4 zum Nachweis der Äquivalenz zweier Spezifikationen gegeben.

Die Trennung von Daten- und Prozeßsprache erlaubt eine Verifikation einer Spezifikation in zwei Schritten. Im ersten Schritt werden die funktionalen Datenausdrücke mit den für funktionale Sprachen existierenden Standardverfahren verifiziert. Anschließend werden die Prozeßelemente der Spezifikation überprüft, z.B., wie hier gezeigt, durch die Anwendung der axiomatischen Theorie.

Im folgenden vergleichen wir $\mathcal{P}_\lambda$ mit verwandten Ansätzen, die ebenfalls parallele Prozesse mit funktionaler Datenbeschreibung verbinden.

ProFun. Die Sprache *ProFun* [Geh96, GH96] des Autors verbindet wie der $\mathcal{P}_\lambda$-Kalkül eine prozeßorientierte Verhaltenssprache mit funktionaler Datenbehandlung. Im Gegensatz zu $\mathcal{P}$ ist die Datensprache festgelegt; es handelt sich um eine an *Standard ML* [MTH90] orientierte Sprache, die Funktionen höherer Ordnung, *pattern matching*, die Behandlung von Ausnahmen (*exception handling*) sowie die Definition abstrakter Datentypen unterstützt. Die Prozeßsprache enthält die Sprachelemente für Prozeßerzeugung, sequentielle Komposition sowie verschiedene Auswahloperatoren für die Auswahl in Abhängigkeit von Daten bzw. durch Kommunikation mit der Umwelt. Eine Behandlung von Ausnahmen der Datenteilsprache ist möglich. In [Geh96] ist die in $\mathcal{P}_\lambda$ vorhandene Mobilität nicht vorgesehen; die Kanalmenge eines Programms ist statisch und es ist nicht möglich, Kanäle durch Interaktion zu übertragen. *ProFun* wurde in einer zweiten Version der Sprache um Mobilität erweitert [GH96]: Wie in $\mathcal{P}$ sind Kanäle Werte

eines speziellen Datentyps und können als Parameter in Nachrichten übertragen werden.

Für beide Versionen der Sprache existieren Compiler-Prototypen, die Programme nach *C++* [Str92] übersetzen und somit die Ausführung von Spezifikationen ermöglichen [Geh96, Fir97].

In [Geh96] wurde eine operationelle Semantik für *ProFun* angegeben. Sie verwendet wie der *Fork*-Kalkül (siehe Seite 32) eine zweistufige Semantik: Die erste Stufe beschreibt die lokalen Aktionen eines einzigen Prozesses, die zweite Stufe beschreibt das Verhalten des Gesamtsystems. Die Zustände (t, env) des lokalen Transitionssystems enthalten neben dem aktuellen Term t eine *Umgebung* (*environment*), die die freien Bezeichner von t mit Werten belegt. Die Zustände des globalen Transitionssystems bestehen aus sogenannten *Prozeßpools* $[p_1 \mapsto (t_1, env_1), ..., p_n \mapsto (t_n, env_n)]$, die für jeden aktiven Prozeß mit dem Prozeßbezeichner p_i[1] den aktuellen lokalen Zustand beinhalten. Aufgrund der komplexen Sprachelemente für die Prozeßbehandlung ist die Spezifikation der Semantik für *ProFun* trotz der Trennung zwischen Daten- und Verhaltenssprache sehr aufwendig. Außerdem ist die Semantik durch die Angabe der Umgebungen in den lokalen Zuständen nicht kompositionell. Daher wurden weder Äquivalenzen auf *ProFun*-Programmen untersucht noch eine axiomatische Theorie angegeben.

Der Kalkül $\mathcal{P}$ enthält im Verhältnis zur Prozeßteilsprache von *ProFun* orthogonalere Sprachelemente, da er sich auf wenige Operatoren beschränkt. So lassen sich in $\mathcal{P}$ sowohl die Auswahl in Abhängigkeit von Daten als $[E_1]; t_1 + ... + [E_n]; t_n$ als auch die Auswahl in Abhängigkeit von Kommunikation als $E_1'?x_1; t_1' + ... + E_m'?x_m; t_m'$ durch den Auswahloperator + realisieren. Für diese beiden Konzepte existieren in *ProFun* zwei verschiedene Operatoren. Weiterhin lassen sich in $\mathcal{P}$ beliebige Terme mit *spawn* als Prozeß starten, während in *ProFun* nur die Prozesse aus Θ als eigene Prozesse ausführbar sind. $\mathcal{P}$ enthält hingegen keine Möglichkeiten zur Behandlung von Ausnahmen. Es bleibt zu untersuchen, inwieweit sich die komplexen Sprachoperatoren von *ProFun* durch die einfacheren Operatoren von $\mathcal{P}$ darstellen lassen. Eine Übersetzung einer Teilmenge von *ProFun* in eine Vorversion von $\mathcal{P}$ wurde in [Fir98] durchgeführt.

Extended LOTOS. Die Sprache *Extended LOTOS* (E-LOTOS) [ISO97] ist eine neue Version der von der ISO standardisierten Spezifikationssprache LOTOS [BB87, ISO87]. Während LOTOS die Beschreibung von Daten in der algebraischen Sprache *ACT ONE* [Man88a, Man88b] vorsieht, verwendet E-LOTOS zu diesem Zweck eine funktionale Sprache. Im Gegensatz zu $\mathcal{P}_\lambda$ ist sind Prozeß- und Datensprache nicht voneinander getrennt, sondern die Datensprache ist Bestandteil der Prozeßsprache. Funktionen werden in Form von Prozessen realisiert, die bestimmte Eigenschaften erfüllen müssen: Sie dürfen nicht mit anderen Prozessen

[1] Hierbei handelt es sich um Bezeichner, die dynamisch den erzeugten Prozessen zugeordnet werden. Es handelt sich hierbei nicht um die Prozeßnamen aus der Prozeßumgebung Θ.

kommunizieren und dürfen weder Echtzeitanforderungen noch Nichtdeterminismus enthalten. Durch die Realierung als Prozesse sind Funktionen keine Werte, so daß E-LOTOS, im Gegensatz zu $\mathcal{P}_\lambda$, nicht die Verwendung von Funktionen höherer Ordnung erlaubt. Daher können viele Standardverfahren der Funktionalen Programmierung [Rea89, Thi94] wie z.B. die Übergabe von Funktionen als Argumente an andere Funktionen nicht angewendet werden. Ein Beispiel für eine solche Übergabe ist die Übergabe der Ordnungsrelation als Parameter an eine Sortierfunktion.

Durch enge Verzahnung von Daten- und Prozeßsprache lassen sich in E-LOTOS imperative Sprachkonstrukte wie z.B. Schleifen simulieren. Andererseits wird die formale Semantik sehr aufwendig [ISO97], was die Anwendung von Standard-Verifikationsverfahren erschwert. Außerdem ist die Integration verschiedener Datensprachen in die Prozeßsprache nicht vorgesehen.

Concurrent ML. Die Sprache *Concurrent ML* (CML) von Reppy [Rep89, Rep92, Rep99] erweitert die funktionale Sprache *Standard ML* [MTH90, Pau91] um Sprachelemente für parallele Prozesse. Hierbei wird nicht zwischen Daten- und Prozeßteilsprache unterschieden, sondern die Sprachkonstrukte für Kommunikation und Prozeßerzeugung sind Funktionen, die beliebig in Ausdrücken verwendet werden können. Zudem sind Kommunikationsaktionen Werte eines speziellen polymorphen Datentyps *event*, die erst durch Anwendung einer speziellen Funktion aktiviert werden. Hieraus ergibt sich eine große Ausdrucksfähigkeit der Sprache, da z.B. Aktionen in Datenstrukturen abgelegt bzw. als Argumente oder Ergebnisse in Funktionsaufrufen verwendet werden können.

Durch die fehlende Unterteilung in Daten- und Prozeßsprache sind Semantiken für CML in der Regel sehr komplex [FH95, FHJ95, Jef96, NN96], insbesondere wenn sowohl Daten- als auch Verhaltensaspekte berücksichtigt werden sollen. So wird z.B. in dem auf den ersten Blick trivial erscheinenden Ausdruck

$$\text{let } x = \mathit{spawn}(...) \text{ in } 3$$

ein Bezeichner x definiert, an den der Wert eines Ausdrucks gebunden wird, der einen Prozeß erzeugt. Da x zur Berechnung des *let*-Ausdrucks nicht verwendet wird, wird die Berechnung des Ausdrucks für x in Analyseverfahren normalerweise nicht berücksichtigt. Da dieser Ausdruck aber einen Prozeß erzeugt, kann er durch Interaktion andere Teile des Programmes beeinflussen. Auf diese Weise wird die Analyse von Programmen durch die symmetrische Integration von Prozeß- und Datenteilsprache aufwendiger, da die existierenden Verfahren zur Analyse und Verifikation von funktionalen Programmen um die Behandlung nichtlokaler Effekte wie z.B. Kommunikation erweitert werden müssen.

Weiterhin ist die Standard-Bisimulation für CML nicht ausreichend, da die Beschriftungen der Transitionen Operatoren für die Verhaltensbeschreibung ent-

halten können. Die zwei folgenden Aktionen

$$c!(fn\ x \Rightarrow 3 + x) \text{ und}$$

$$c!(fn\ x \Rightarrow \mathbf{ch}\ d.\ spawn(d!x + 3); d?y; y)$$

senden Funktionen als Werte über den Kanal c. Während die erste Funktion einen einfachen Ausdruck zur Berechnung des Wertes verwendet, enthält die zweite Funktion eine Interaktion mit einem lokalen Prozeß über einen lokalen Kanal. Um zu entscheiden, ob die beiden Aktionen äquivalentes Verhalten ausweisen, muß zuvor die Äquivalenz der Transitionsbeschriftungen, d.h. der zu übertragenden Funktionen, analysiert werden. Daher ist die Verwendung einer *Bisimulation höherer Ordnung* (*higher-order bisimulation*) [Tho95] notwendig, die neben den Zuständen des Transitionssystems auch die Transitionsbeschriftungen in Beziehung setzt [FHJ95]. Aus diesem Grund führt die durch die Mischung von Prozeß- und Datensprache erzielte hohe Ausdrucksfähigkeit von CML zu einer komplexen Semantik und aufwendigen Äquivalenzbetrachtungen.

Durch die asymmytrische Integration von Daten- und Verhaltensteilsprache in $\mathcal{P}_\lambda$ ist die Auswertung von funktionalen Ausdrücken wie in anderen funktionalen Sprachen durch eine Reduktionssemantik ohne die Berücksichtigung von Parallelität und Seiteneffekten möglich. Daher ist die Semantik von $\mathcal{P}_\lambda$ im Verhältnis zu Semantiken für CML weniger aufwendig. Außerdem können Verhaltensaspekte nicht in Transitionsbeschriftungen auftreten, so daß auf die Verwendung einer Bisimulation höherer Ordnung verzichtet werden kann.

Facile. Wie CML ist *Facile* [GMP89, TLP+93] von Prasad et al. eine Erweiterung der funktionalen Sprache *Standard ML* [MTH90, Pau91] um Sprachelemente für Kommunikation und Prozeßbehandlung. Auch in diesem Ansatz können Prozeß- und Datensprache beliebig kombiniert werden; im Gegensatz zu CML sind Kommunikationsaktionen aber keine Datenwerte. Über CML hinausgehend ist das Konzept der verteilten Programme, das es ermöglicht, die Prozesse eines Programms auf verschiedenen *Lokalitäten* verteilt (z.B. auf den Knoten eines Rechnernetzwerks) ablaufen zu lassen.

In [TLG92] wird eine operationalle Semantik für eine Teilmenge der Sprache durch die Angabe von Transitionsregeln definiert. Zur Darstellung der Semantik von verteilten Programmen werden zusätzliche Regeln für das Erzeugen neuer Lokalitäten und das Starten von Prozessen auf Lokalitäten definiert. Hierzu enthalten die Zustände eines Transitionssystems neben dem jeweiligen Programmterm auch die Mengen der bisher erzeugten Kanäle und der Lokalitäten. Allerdings existiert für diese Semantik keine axiomatische Theorie.

In [Ama94] wird ebenfalls eine Teilmenge von *Facile*, genannt *Core Facile*, untersucht. Sie besteht aus einem um Synchronisationsprimitive und Kanalerzeugung erweiterten λ-Kalkül. Für diese Sprache wird eine Reduktionssemantik definiert, deren Reduktionsschritte durch unbeschriftete Transitionen dargestellt

werden. Da Vergleiche von Transitionsbeschriftungen nicht möglich sind, wird als Äquivalenz auf Ausdrücken die *barbed bisimulation* [MS92] verwendet: Zwei Terme sind äquivalent, wenn sie äquivalente Reduktionsschritte ausführen können und in der Interaktion mit ihrer Umwelt dieselbe Folge von Kommunikationsaktionen ausführen. Allerdings fehlt eine Axiomatisierung dieser Äquivalenz. Weiterhin wird eine asynchrone Variante von *Core Facile* angegeben, für die eine Semantik in Form einer abstrakten Zustandsmaschine definiert wird.

4.5 Beweise

Theorem 4.2. Wenn $\vdash E : T$ und $E \hookrightarrow E'$, dann $\vdash E' : T$.

Für den Beweis von Theorem 4.2 benötigen wir das folgende Hilfslemma:

Lemma 4.6 *Wenn* $x : T' \vdash E : T$ *und* $\vdash v : T'$*, dann gilt* $\vdash E\langle v/x\rangle : T$.

Beweis für Lemma 4.6. Analog zum Beweis von Lemma 3.19. □

Beweis für Theorem 4.2. Wir nehmen an, daß $\vdash E : T$ sowie $E \hookrightarrow E'$ gilt und beweisen das Theorem für die Regeln der Reduktionssemantik in Abbildung 4.3.

$\mathbf{R}_{17}$ Dann gilt $E = \mathit{if}\ E_1\ \mathit{then}\ E_2\ \mathit{else}\ E_3$, $E_1 \hookrightarrow E_1'$ und $E' = \mathit{if}\ E_1'\ \mathit{then}\ E_2\ \mathit{else}\ E_3$. Mit T_{21} wissen wir, daß $\vdash E_1 : \mathit{bool}$, $\vdash E_2 : T$ und $\vdash E_3 : T$ gilt. Mit der Induktionshypothese können wir schließen, daß $\vdash E_1' : \mathit{bool}$. Durch Anwendung von T_{21} erhalten wir $\vdash \mathit{if}\ E_1'\ \mathit{then}\ E_2\ \mathit{else}\ E_3 : T$.

$\mathbf{R}_{18}$ Dann gilt $E = \mathit{if}\ \mathit{true}\ \mathit{then}\ E_1\ \mathit{else}\ E_2$ und $E' = E_1$. Durch Anwendung von T_{21} erhalten wir $\vdash E_1 : T$.

$\mathbf{R}_{19}$ Dann gilt $E = \mathit{if}\ \mathit{false}\ \mathit{then}\ E_1\ \mathit{else}\ E_2$ und $E' = E_2$. Mit T_{21} wissen wir, daß $\vdash E_2 : T$ gilt. Mit der Induktionshypothese erhalten wir $\vdash E_1' : T\ \mathit{list}$ und folglich $\vdash \mathit{head}\ E_1' : T$.

$\mathbf{R}_{20}$ Dann gilt $E = (V_1, ..., V_{i-1}, E_i, ..., E_n)$ und $E' = (V_1, ..., V_{i-1}, E_i', ..., E_n)$. Mit T_{22} wissen wir, daß $\exists T_i :\vdash E_i : T_i$ gilt. Dann können wir mit der Induktionshypothese schließen, daß $\vdash E_i' : T_i$ und folglich $\vdash E' : T$ gilt.

$\mathbf{R}_{21}$ Dann gilt $E = \#i\ (V_1, ..., V_n)$ und $E' = V_i$. Mit T_{22}, T_{23} wissen wir, daß $\exists T_1, ..., T_n : T = T_1 \times ... \times T_n$, $\vdash V_j : T_j$ für alle $1 \leq j \leq n$ und $\vdash \#i\ (V_1, ..., V_n) : T_i$. Daher wissen wir, daß $\vdash V_i : T_i$ gilt.

$\mathbf{R}_{22}$ Dann gilt $E = \mathit{cons}\ E_1\ E_2$, $E_1 \hookrightarrow E_1'$ und $E' = \mathit{cons}\ E_1'\ E_2$. Mit T_{25} wissen wir, daß $\vdash \mathit{cons}\ E_1\ E_2 : T\ \mathit{list}$ und $\vdash E_1 : T$ gilt. Durch Anwendung der Induktionshypothese erhalten wir $\vdash E_1' : T$ und folglich $\vdash E' : T\ \mathit{list}$.

$\mathbf{R}_{23}$ Dann gilt $E = \mathit{cons}\ V\ E_2$, $E_2 \hookrightarrow \mathit{cons}\ V\ E_2'$ und $E' = \mathit{cons}\ V\ E_2'$. Mit T_{25} wissen wir, daß $\vdash E : T\ \mathit{list}$ und $\vdash E_2 : T\ \mathit{list}$ gilt. Durch Anwendung der Induktionshypothese erhalten wir $\vdash E_2' : T\ \mathit{list}$ und folglich $\vdash E' : T\ \mathit{list}$.

$\mathbf{R}_{24}$ Dann gilt $E = \mathit{head}\ E_1$, $E_1 \hookrightarrow E_1'$ und $E' = \mathit{head}\ E_1'$. Mit T_{26} können wir folgern, daß $\vdash \mathit{head}\ E_1 : T$ und $\vdash E_1 : T\ \mathit{list}$ gilt.

$\mathbf{R}_{25}$ Dann gilt $E = \mathit{head}\ (\mathit{cons}\ V_1\ V_2)$ und $E' = V_1$. Mit T_{26}, T_{25} wissen wir, daß $\vdash E : T$, $\vdash \mathit{cons}\ V_1\ V_2 : T\ \mathit{list}$ und $\vdash x : T$ gilt. Daher wissen wir, daß $\vdash E' : T$ gilt.

$\mathbf{R}_{26}$ Dann gilt $E = \mathit{tail}\, E_1$, $E_1 \hookrightarrow E_1'$ und $E' = \mathit{tail}\, E_1'$. Mit T_{27} wissen wir, daß $\vdash E : T\,\mathit{list}$ und $\vdash E_1 : T\,\mathit{list}$ gilt. Durch Anwendung der Induktionshypothese erhalten wir $\vdash E_1' : T\ \mathit{list}$ und folglich $\vdash E' : T\ \mathit{list}$.

$\mathbf{R}_{27}$ Dann gilt $E = \mathit{tail}\ (\mathit{cons}\ V_1\ V_2)$ und $E' = V_2$. Mit T_{27} wissen wir, daß $\vdash E : T\ \mathit{list}$, $\vdash \mathit{cons}\ V_1\ V_2 : T\ \mathit{list}$ und $\vdash V_2 : T\ \mathit{list}$ gilt. Daher wissen wir, daß $\vdash E' : T\ \mathit{list}$.

$\mathbf{R}_{28}$ Dann gilt $E = E_1\ E_2$, $E_1 \hookrightarrow E_1'$ und $E' = E_1'\ E_2$. Mit T_{29} wissen wir, daß $\vdash E_1 : T' \to T$ und $\vdash E_2 : T'$ gilt. Durch Anwendung der Induktionshypothese erhalten wir $\vdash E_1' : T' \to T$. Daher können wir mit T_{29} folgern, daß $\vdash E_1' E_2 : T$ gilt.

$\mathbf{R}_{29}$ Dann gilt $E = V\ F$, $F \hookrightarrow F'$ und $E' = V\ F'$. Mit T_{29} wissen wir, daß $\vdash V : T' \to T$ und $\vdash F : T'$ gilt. Durch Anwendung der Induktionshypothese erhalten wir $\vdash F' : T'$ und folglich $\vdash V\ F' : T$ gilt.

$\mathbf{R}_{30}$ Dann gilt $E = (\lambda x.F)\, V$ und $E' = F\langle v/x\rangle$. Mit T_{29} wissen wir, daß $x : T' \vdash F : T' \to T$ und $\vdash V : T'$ gilt. Dann können wir mit Lemma 4.6 und T_{13} folgern, daß $\vdash F\langle V/x\rangle : T$.

$\mathbf{R}_{31}$ Dann gilt $E = (\mathrm{let}\ \ x = E_1\ \ \mathrm{in}\ \ E_2)$, $E_1 \hookrightarrow E_1'$ und $E' = (\mathrm{let}\ \ x = E_1'\ \ \mathrm{in}\ \ E_2)$. Mit T_{30} wissen wir, daß $\vdash E_1 : T'$ und $x : T' \vdash E_2 : T$. Durch Anwendung der Induktionshypothese erhalten wir in Verbindung mit T_{30} $\vdash E_1' : T'$ und folglich $\vdash \mathrm{let}\ \ x = E_1'\ \ \mathrm{in}\ \ E_2 : T$.

$\mathbf{R}_{32}$ Dann gilt $E = (\mathrm{let}\ \ x = V\ \ \mathrm{in}\ \ F)$ und $E' = F\langle v/x\rangle$. Mit T_{30} wissen wir, daß $\vdash V : T'$ und $x : T' \vdash F : T$ gilt. Dann können wir mit Lemma 4.6 und T_{13} folgern, daß $\vdash F\langle v/x\rangle : T$ gilt.

$\mathbf{R}_{33}$ Dann gilt $E = Y\, E_1\, E_2$, $E_1 \hookrightarrow E_1'$ und $E' = Y\, E_1'\, E_2$. Mit T_{31} wissen wir, daß $\vdash E : T'$, $\vdash E_1 : (T \to T') \to T \to T'$ und $\vdash E_2 : T$ gilt. Durch Anwendung der Induktionshypothese erhalten wir $\vdash E_1' : (T \to T') \to T \to T'$ und folglich $\vdash E' : T'$.

R$_{34}$ Dann gilt $E = Y\ V\ E_2$, $E_2 \hookrightarrow E_2'$ und $E' = Y\ V\ E_2'$. Mit T_{31} wissen wir, daß $\vdash E : T'$, $\vdash V : (T \to T') \to T \to T'$ und $\vdash E_2 : T$ gilt. Durch Anwendung der Induktionshypothese erhalten wir $\vdash E_2' : T$ und folglich $\vdash E' : T'$.

R$_{35}$ Dann gilt $E = Y\ V_1\ V_2$ und $E' = (V_1\ (Y\ V_1))\ V_2$. Mit T_{31} wissen wir, daß $\vdash E : T'$, $\vdash V_1 : (T \to T') \to T \to T'$ und $\vdash V_2 : T$ gilt. Hieraus können wir $\vdash Y\ V_1 : T \to T'$ und $\vdash V_1\ (Y\ V_1) : T \to T'$ schließen. Daher wissen wir, daß $\vdash E' : T'$ gilt.

□

Theorem 4.5.

Die Systeme

$$\mathbf{ch}\ rc.\ spawn(RPC\_Simulate(rc)); \mathbf{ch}\ result.\ rc!(write, adr, v, result); result?x; \mathbf{1}$$

und

$$\mathbf{ch}\ rc.\ spawn(RPC\_Server(rc)); \mathbf{ch}\ result.\ rc!(write, adr, v, result); result?x; \mathbf{1}$$

zeigen bzgl. der schwachen Kongruenzrelation äquivalentes Verhalten.

Beweis für Theorem 4.5. Für den Beweis nehmen wir an, daß die Speicherkomponente korrekt arbeitet, d.h. nach dem Senden eines Tupels (adr, v, c) auf *read* bzw. *write* wird das Ergebnis der Anfrage auf dem Kanal c gesendet.

Zunächst wenden wir die Axiome der Theorie $\mathcal{AX}_{\simeq_\beta}$ auf das System ohne RPC-Komponente an. Dieses System wird durch einen Term beschrieben, in dem zuerst eine Instanz von *RPC_Simulate* generiert wird. Danach wird ein lokaler Kanal *result* erzeugt, der als Rückgabekanal während des Schreibzugriffs auf die Speicherkomponente fungiert. Die Gleichungen, die in den einzelnen Umformungsschritten verwendet werden, werden in Form von Kommentaren angegeben.

$$\begin{array}{l}
\quad \mathbf{ch}\ rc. \\
\quad\quad spawn(RPC_Simulate(rc)); \mathbf{ch}\ result.\ rc!(write, adr, v, result); result?x; \mathbf{1} \\
\quad\quad\quad // \text{ Ax. (3.25)} \\
= \mathbf{ch}\ rc. \\
\quad spawn(\tau; rc?req; \\
\quad\quad \{proc_ch \leftarrow \#1\ req, adr \leftarrow \#2\ req, value \leftarrow \#3\ req, return_ch \leftarrow \#4\ req\}; \\
\quad\quad spawn(proc_ch!(adr, value, return_ch)); RPC_Simulate(rc)); \\
\quad \mathbf{ch}\ result.\ rc!(write, adr, v, result); result?x; \mathbf{1} \\
\quad\quad\quad // \text{ Ax. (3.24)} \\
= \mathbf{ch}\ rc, result. \\
\quad\quad spawn(\tau; rc?req;
\end{array}$$

$\{proc\_ch \leftarrow \#1\, req, adr \leftarrow \#2\, req, value \leftarrow \#3\, req, return\_ch \leftarrow \#4\, req\};$
$spawn(proc\_ch!(adr, value, return\_ch)); RPC\_Simulate(rc));$
$rc!(write, adr, v, result); result?x; \mathbf{1}$
// Ax. (3.36)
$= \mathbf{ch}\ rc, result.$
$\tau; spawn(rc?req;$
$\{proc\_ch \leftarrow \#1\, req, adr \leftarrow \#2\, req, value \leftarrow \#3\, req, return\_ch \leftarrow \#4\, req\};$
$spawn(proc\_ch!(adr, value, return\_ch)); RPC\_Simulate(rc));$
$rc!(write, adr, v, result); result?x; \mathbf{1}$
$+rc!(write, adr, v, result);$
$spawn(\tau; rc?req;$
$\{proc\_ch \leftarrow \#1\, req, adr \leftarrow \#2\, req, value \leftarrow \#3\, req, return\_ch \leftarrow \#4\, req\};$
$spawn(proc\_ch!(adr, value, return\_ch)); RPC\_Simulate(rc));$
$result?x; \mathbf{1}$
// Ax. (3.36), rc lokal, Ax. (3.18), (3.20), (3.8)
$= \mathbf{ch}\ rc, result.$
$\tau; \tau; spawn(spawn(write!(adr, v, result)); RPC\_Simulate(rc)); result?x; \mathbf{1}$
// Ax. (3.28)
$= \mathbf{ch}\ rc, result.$
$\tau; \tau; spawn(write!(adr, v, result)); spawn(RPC\_Simulate(rc)); result?x; \mathbf{1}$
// Ax. (3.37)
$= \mathbf{ch}\ rc, result.$
$\tau; spawn(write!(adr, v, result)); spawn(RPC\_Simulate(rc)); result?x; \mathbf{1}$
// Ax. (3.27)
$= \mathbf{ch}\ rc, result.\ \tau;$
$spawn(RPC\_Simulate(rc)); spawn(write!(adr, v, result)); result?x; \mathbf{1}$
// Ax. (3.4), (3.36), $result$ lokal, Ax. (3.18), (3.20), (3.8)
$= \mathbf{ch}\ rc, result.$
$\tau; spawn(RPC\_Simulate(rc)); write!(adr, v, result); spawn(\mathbf{1}); result?x; \mathbf{1}$
// $spawn(\mathbf{1}) = \mathbf{1}$, Ax. (3.3)
$= \mathbf{ch}\ rc, result.\ \tau; spawn(RPC\_Simulate(rc)); write!(adr, v, result); result?x; \mathbf{1}$

Nun wenden wir die Theorie $\mathcal{AX}_{\simeq_\beta}$ auf das System mit RPC-Komponente an.

$\mathbf{ch}\ rc.\ spawn(RPC\_Server(rc)); \mathbf{ch}\ result.\ rc!(write, adr, v, result); result?x; \mathbf{1}$
// Ax. (3.25)
$= \mathbf{ch}\ rc.$
$spawn(\tau; rc?req; spawn(RPC\_Client(req)); RPC\_Server(rc));$
$\mathbf{ch}\ result.\ rc!(write, adr, v, result); result?x; \mathbf{1}$
// Ax. (3.24)
$= \mathbf{ch}\ rc, result.$
$spawn(\tau; rc?req; spawn(RPC\_Client(req)); RPC\_Server(rc));$

$$
\begin{aligned}
& rc!(write, adr, v, result); result?x; \mathbf{1} \\
& \quad // \text{ Ax. (3.36)} \\
= \ & \mathbf{ch}\ rc, result. \\
& \quad \tau; spawn(rc?req; spawn(RPC\_Client(req)); RPC\_Server(rc)); \\
& \quad\quad rc!(write, adr, v, result); result?x; \mathbf{1} \\
& \quad + rc!(write, adr, v, result); \\
& \quad\quad spawn(\tau; rc?req; spawn(RPC\_Client(req)); RPC\_Server(rc)); \\
& \quad\quad result?x; \mathbf{1} \\
& \quad\quad // \ rc \text{ lokal, Ax. (3.18), (3.20), (3.8)} \\
= \ & \mathbf{ch}\ rc, result. \\
& \quad \tau; spawn(rc?req; spawn(RPC\_Client(req)); RPC\_Server(rc)); \\
& \quad\quad rc!(write, adr, v, result); result?x; \mathbf{1} \\
& \quad\quad // \text{ Ax. (3.36)} \\
= \ & \mathbf{ch}\ rc, result.\ \tau; \\
& \quad (rc?req; spawn(spawn(RPC\_Client(req)); RPC\_Server(rc)); \\
& \quad\quad rc!(write, adr, v, result); result?x; \mathbf{1} \\
& \quad + rc!(write, adr, v, result); \\
& \quad\quad spawn(rc?req; spawn(RPC\_Client(req)); RPC\_Server(rc)); \\
& \quad\quad result?x; \mathbf{1} \\
& \quad + \tau; spawn(spawn(RPC\_Client((write, adr, v, result))); RPC\_Server(rc)); \\
& \quad\quad result?x; \mathbf{1}) \\
& \quad\quad // \ rc \text{ lokal, Ax. (3.18), (3.20), (3.8)} \\
= \ & \mathbf{ch}\ rc, result. \\
& \quad \tau; \tau; \\
& \quad spawn(spawn(RPC\_Client((write, adr, v, result))); RPC\_Server(rc)); \\
& \quad result?x; \mathbf{1} \\
& \quad\quad // \text{ Ax. (3.37)} \\
= \ & \mathbf{ch}\ rc, result. \\
& \quad \tau; \\
& \quad spawn(spawn(RPC\_Client((write, adr, v, result))); RPC\_Server(rc)); \\
& \quad result?x; \mathbf{1} \\
& \quad\quad // \text{ Ax. (3.28)} \\
= \ & \mathbf{ch}\ rc, result. \\
& \quad \tau; \\
& \quad spawn(RPC\_Client((write, adr, v, result))); spawn(RPC\_Server(rc)); \\
& \quad result?x; \mathbf{1} \\
& \quad\quad // \text{ Ax. (3.25)} \\
= \ & \mathbf{ch}\ rc, result. \\
& \quad \tau; \\
& \quad spawn(\tau; \mathbf{ch}\ local\_ch.\ write!(adr, v, local\_ch); local\_ch?x; result!x); \\
& \quad\quad spawn(RPC\_Server(rc)); result?x; \mathbf{1} \\
& \quad\quad // \text{ Ax. (3.38), (3.37)} \\
= \ & \mathbf{ch}\ rc, result.
\end{aligned}
$$

$\tau; spawn(\mathbf{ch}\ local\_ch.\ write!(adr, v, local\_ch); local\_ch?x; result!x);$
$spawn(RPC\_Server(rc)); result?x; \mathbf{1}$
// Ax. (3.27)
$= \mathbf{ch}\ rc, result.$
$\tau; spawn(RPC\_Server(rc))$
$spawn(\mathbf{ch}\ local\_ch.\ write!(adr, v, local\_ch); local\_ch?x; result!x);$
$result?x; \mathbf{1}$
// Ax. (3.24),(3.36), rc lokal, Ax. (3.18), (3.20), (3.8)
$= \mathbf{ch}\ rc, result, local\_ch.$
$\tau; spawn(RPC\_Server(rc));$
$write!(adr, v, local\_ch); spawn(local\_ch?x; result!x); result?x; \mathbf{1}$
// Ax. (3.36), rc lokal, Ax. (3.18), (3.20), (3.8)
$= \mathbf{ch}\ rc, result, local\_ch.$
$\tau; spawn(RPC\_Server(rc));$
$write!(adr, v, local\_ch); local\_ch?x; spawn(result!x); result?x; \mathbf{1}$
// Ax. (3.36), rc lokal,Ax. (3.18), (3.20), (3.8)
$= \mathbf{ch}\ rc, result, local\_ch.\ \tau; spawn(RPC\_Server(rc)); write!(adr, v, local\_ch);$
$local\_ch?x; \tau; \mathbf{1}$
// Ax. (3.38), (3.40), (3.16)
$= \mathbf{ch}\ rc, local\_ch.\ \tau; spawn(RPC\_Server(rc)); write!(adr, v, local\_ch);$
$local\_ch?x; \mathbf{1}$
// α-conversion
$= \mathbf{ch}\ rc, result.\ \tau; spawn(RPC\_Server(rc)); write!(adr, v, result); result?x; \mathbf{1}$

Es zeigt sich, daß beide Systeme durch das Anwenden der Gleichungen in Terme umgewandelt werden können, die sich nur durch den Aufruf von *RPC_Simulate* bzw. *RPC_Server* unterscheiden. Beide Terme können keine weiteren Aktionen durchführen, da sowohl *RPC_Simulate* als auch *RPC_Server* eine Eingabe auf dem restringierten Kanal rc erwarten. Daher haben wir gezeigt, daß die beiden Systeme bzgl. der schwachen Kongruenz äquivalentes Verhalten zeigen. □

Kapitel 5

Diagramme für n-Agenten-Systeme

Nachdem wir in den vorherigen Kapiteln eine textuelle Notation für die Spezifikation von reaktiven Systemen mit Daten vorgestellt haben, führen wir in diesem Kapitel Diagramme zur graphischen Darstellung reaktiver Systemen mit Daten ein. Wir beschränken uns hierbei auf die Spezifikation von Systemen, die aus einer festen Anzahl nebenläufiger Komponenten bestehen. Diese Systeme nennen wir *n-Agenten-Systeme*. Wir geben für diese Systeme eine graphische Darstellung an, deren syntaktische Elemente auf den *Sequenzdiagrammen* der *Unified Modeling Language* (UML) [BRJ98, RJB98] basieren, die aber für die Beschreibung der n-Agenten-Systeme angepaßt wurden. Diese Diagramme nennen wir n-Agenten-Diagramme. Da in existierenden Notationen wie den dynamischen Modellen von UML und den Message Sequence Charts [ITU96a, Ren99] die Daten nur informell behandelt werden, wird für die n-Agenten-Diagramme eine formale Datenintegration entwickelt. Hierzu gehört die Definition von aus Programmiersprachen bekannten Konzepten wie z.B. den Gültigkeitsbereichen von Bezeichnern. Weiterhin erlauben wir, wie für den Kalkül $\mathcal{P}$ aus Kapitel 3, die Integration verschiedener Datensprachen. Die textuellen Datenbeschreibungen werden in Form von speziellen graphischen Elementen in die Verhaltensbeschreibung integriert.

Die n-Agenten-Diagramme sollen zur Erstellung *generativer* Spezifikationen verwendet werden können, die eine Menge möglicher Systemabläufe definieren. Zu diesem Zweck enthält die Notation komplexe Sprachkonstrukte wie die Behandlung von Alternativen sowie Schleifen. Die vergleichbaren Konzepte von UML machen keine Annahmen über ein spezifisches Systemmodell der zu modellierenden Systeme, da sie sowohl zur Spezifikation nebenläufiger als auch prozeduraler Systeme verwendbar sein sollen. Dies führt bei einigen Diagrammelementen von UML zu einer mehrdeutigen Semantik [GGW98, GGW99]. Daher verwenden wir eigene Konstrukte, die speziell für die Modellierung verteilter Systeme geeignet sind und eine eindeutige Semantik besitzen.

Die Grund für die Orientierung unserer Syntax an der Syntax der Sequenzdia-

gramme aus UML ist die gewünschte weitgehende Kompatibilität zu existierenden Werkzeugen wie z.B. Zeichenprogrammen, die die Verwendung der UML-Notation unterstützen. Durch die weite Verbreitung von UML als Notation für die Modellierung von Systemen ist die Entwicklung einer Reihe von Werkzeugen zu erwarten, die die Elemente dieser Notation beinhalten. Bei Definition einer vollständig eigenen Syntax würde die Einsetzbarkeit dieser Werkzeuge nicht gegeben sein, so daß die gesamte notwendige Werkzeugunterstützung durch Eigenentwicklungen zu realisieren wäre. Die Semantik der n-Agenten-Diagramme unterscheidet sich von der Semantik der Sequenzdiagramme aus UML, da sie im Gegensatz zur Semantik von UML die Berechung des von den Diagrammen beschriebenen Verhaltens erlaubt, während das Metamodell von UML [BRJ98, OMG99] hierüber keine Aussagen macht.

In Abschnitt 5.1 beschreiben wir zunächst die generellen Eigenschaften von Interaktionsdiagrammen. In Abschnitt 5.2 definieren wir die grundlegenden Annahmen der n-Agenten-Systeme. Danach werden in Abschnitt 5.3 die n-Agenten-Diagramme zur graphischen Spezifikation von n-Agenten-Systemen eingeführt. Anschließend folgt in Abschnitt 5.4 eine textuelle Notation für die n-Agenten-Diagramme, die im folgenden Kapitel 6 als Grundlage für die Angabe einer formalen Semantik fungiert.

5.1 Interaktionsdiagramme

In Systemen, die aus nebenläufigen Komponenten bestehen, treten zwei Arten von Parallelität auf: Mit *Inter-Objekt-Parallelität* wird die parallele Ausführung verschiedener Objekte bzw. Komponenten bezeichnet, während sich *Intra-Objekt-Parallelität* auf die parallele Ausführung von Teilkomponenten innerhalb einer Komponente bezieht. In der Unterscheidung dieser beiden Arten von Parallelität besteht nach Harel die *grundlegende Dualität des Systemverhaltens* [Har97].

Eine Möglichkeit zur Beschreibung von Inter-Objekt-Parallelität besteht in der Verwendung von *Interaktionsdiagrammen*. Diese beschreiben graphisch den Nachrichtenfluß zwischen den Systemkomponenten sowie zwischen den Komponenten und der Systemumgebung.

Interaktionsdiagramme beschreiben einen *Kontext* in Verbindung mit einer *Interaktion* [BRJ98, S.207f]. Ein Kontext ist eine Menge von Objekten, genannt *Instanzen*, die durch den Austausch von Nachrichten interagieren. Die dynamische Abfolge von Nachrichten, die zur Erfüllung einer spezifischen Aufgabe ausgetauscht werden, heißt *Interaktion*. Für einen gegebenen Kontext können mehrere Interaktionen in verschiedenen Diagrammen spezifiziert werden.

Zu den Interaktionsdiagrammen gehören die *Sequenz-* und die *Kollaborationsdigramme* aus der *Unified Modeling Language* (UML) [BRJ98, RJB98, OMG99]. Mit den Sequenzdiagrammen eng verwandt sind die *Message Sequence Charts* [ITU96a, ITU99]. Sowohl Sequenz- als auch Kollaborationsdiagramme stellen den

Nachrichtenfluß innerhalb des Systems dar, bieten aber unterschiedliche Sichtweisen auf das darzustellende System. Sequenzdiagramme und Message Sequence Charts betonen den zeitlichen Ablauf des Austausches von Nachrichten, während Kollaborationsdiagramme den Fokus auf die Beziehungen zwischen den Systemkomponenten legen.

Beispiel 5.1 *Als Beispiel für die Verwendung von Interaktionsdiagrammen spezifizieren wir eine kleine Fallstudie [Pae98, GGW98], die Rückgabe eines entliehenen Buchs in einer Bibliothek, als Sequenz- und als Kollaborationsdiagramm. Als Akteure fungieren ein Leser, ein Bibliothekar und das Computersystem der Bibliothek.*

Der obere Teil von Abbildung 5.1 enthält die Spezifikation der Fallstudie mittels eines Sequenzdiagramms. Die drei Akteure werden als Instanzen dargestellt, die parallel aktiv sind. Die Zeit verläuft von oben nach unten. Die ausgetauschten Nachrichten werden durch Pfeile dargestellt, die mit den Namen der Nachrichten und eventuellen Parametern beschriftet werden. Zunächst gibt der Leser das Buch b an den Bibliothekar. Dieser gibt die Daten des Buchs in das Computersystem ein, welches eine Aktualisierung des Status des Buchs vornimmt. Wurde das Buch von einem Kunden reserviert, wird dieser Kunde benachrichtigt. Anschließend erhält der Bibliothekar eine Bestätigung und und bestätigt seinerseits dem Kunden die Beendigung des Vorgangs.

Der untere Teil von Abbildung 5.1 beschreibt die Buchrückgabe in Form eines Kollaborationsdiagramms. Die Akteure werden als aktive Objekte mit eigenem Verhalten dargestellt. Zusätzlich wird das zurückzugebende Buch als passives Objekt in das Diagramm aufgenommen. Die Reihenfolge der Nachrichten wird in diesem Diagramm durch die Angabe von Sequenznummern definiert. Zusätzlich können die Beziehungen zwischen den Objekten angegeben werden. So wird z.B. definiert, daß das Objekt Book *dem Objekt* Librarian *als Parameter b übergeben wurde.*

Während das Sequenzdiagramm eine übersichtliche Darstellung des zeitlichen Nachrichtenflusses erlaubt, muß dieser Kontrollfluß im Kollaborationsdiagramm aus den Sequenznummern abgeleitet werden. Hingegen kann das Kollaborationsdiagramm zusätzliche Informationen über die Beziehungen zwischen den Komponenten enthalten, die im Sequenzdiagramm nicht darstellbar sind (z.B. die Parameter-Beziehung zwischen Librarian *und* Book*).*

In unseren Diagrammen legen wir, analog zu den Sequenzdiagrammen und den Message Sequence Charts, den Schwerpunkt auf den Kontrollfluß und den damit verbundenen Datenfluß. Daher orientiert sich unsere Notation an diesen Diagrammen. Die Beziehungen zwischen den einzelnen Systemkomponenten werden nicht modelliert. Da Kollaborationsdiagramme aufgrund der Darstellung der Beziehungen zwischen Objekten eher für die Modellierung explizit objektorientierter Systeme geeignet sind, werden sie im folgenden nicht weiter betrachtet.

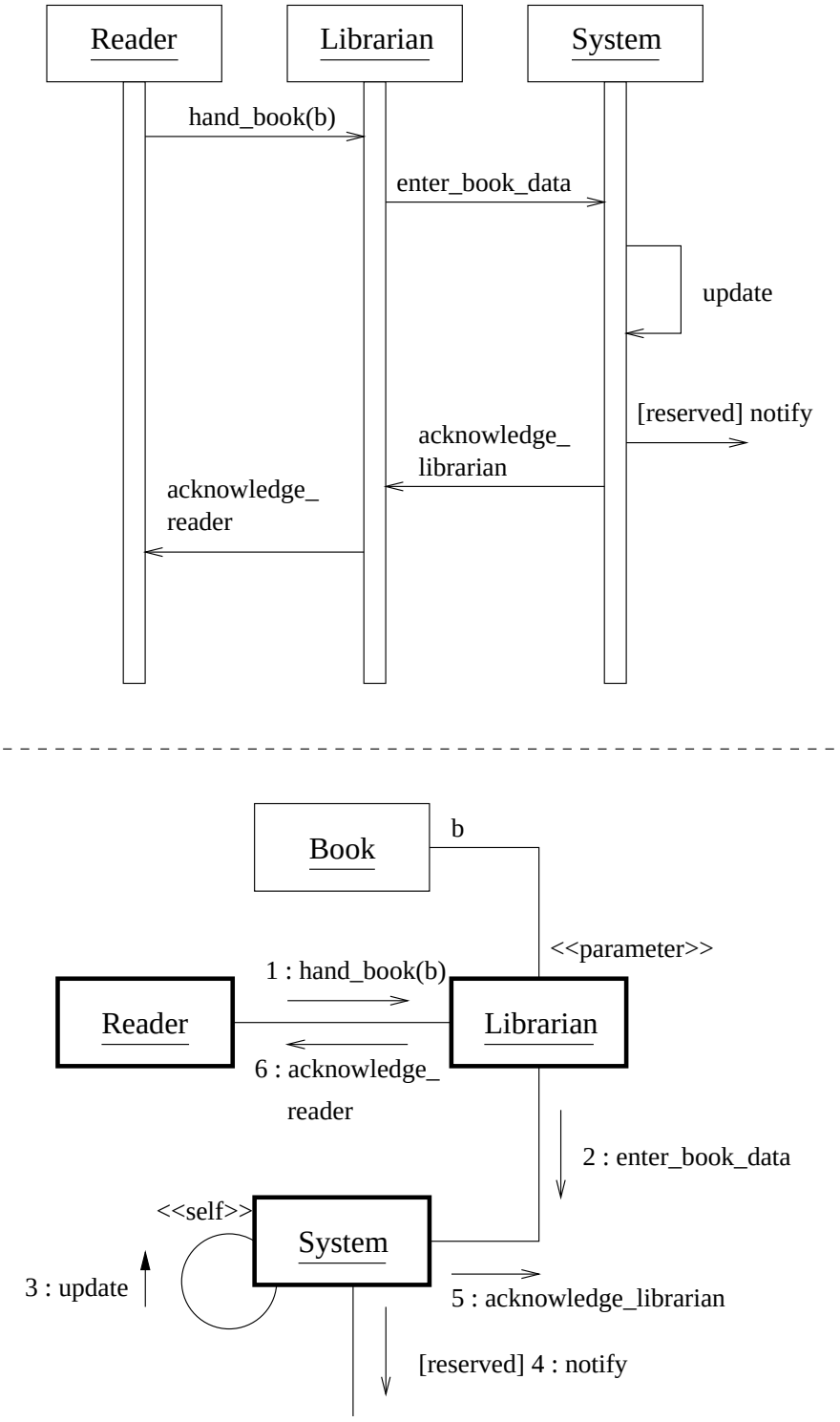


Abbildung 5.1: Beispiele für Interaktionsdiagramme.

5.2 Systemmodell und grundlegende Notation

In diesem Abschnitt erläutern wir das Systemmodell der n-Agenten-Systeme. Wir beschreiben die Systemstruktur sowie die Kommunikation zwischen den Systemteilen. Gleichzeitig geben wir an, welche graphischen Elemente aus den Sequenzdiagrammen von UML zur Darstellung von Systemeigenschaften übernommen werden.

Systemstruktur. Ein System besteht aus einer festen Menge von Instanzen, genannt *Agenten*, die parallel ausgeführt werden. Jede Instanz verfügt über lokalen Speicher; ein globaler Speicher, auf den alle Instanzen zugreifen können, ist nicht vorgesehen. Der Austausch von Informationen findet durch Kommunikation statt. Das System ist in eine *Systemumgebung* eingebunden, mit der ebenfalls ein Datenaustausch durch Kommunikation möglich ist. Diese Umgebung bezeichnen wir mit **env**.

Wie in den Sequenzdiagrammen von UML stellen wir eine Instanz als einen Kasten dar, der den Namen der Instanz beinhaltet. Um anzudeuten, daß es sich hier um eine konkrete Komponente handelt, ist der Name wie in den Sequenzdiagrammen unterstrichen. Die Benennung von Diagrammen muß eindeutig sein, daher dürfen nicht mehrere Diagramme denselben Namen tragen.

In Sequenzdiagrammen wird zwischen den aktiven und passiven Phasen einer Instanz unterschieden. Ist eine Instanz aktiv, wird ihre Lebenslinie durch ein dünnes senkrechtes Rechteck, genannt *focus of control*, dargestellt [BRJ98, S. 247]. Die passiven Phasen sind durch eine gestrichelte Lebenslinie gekennzeichnet. Da in unserem Systemmodell die Instanzen aufgrund der parallelen Ausführung während der gesamten Systemlaufzeit aktiv sind, zeichnen wir die ganze Lebenslinie einer Instanz als Rechteck (siehe Abbildung 5.3).

Tore. Jede Instanz verfügt über eine Menge von *Toren* (*gates*[1]), über die Nachrichten an die Instanz gesendet werden können. Diese Tore fungieren als Eingabeschnittstelle für die Interaktion mit der Instanz, indem sie festlegen, welche Nachrichten an eine Instanz gesendet werden können und von welcher Art die dabei übertragenen Werte sind. Durch die Tore erreichen wir somit eine Typisierung der Nachrichten. Instanzen mit ihren Toren geben wir in Form von Instanzen aus Klassendiagrammen in UML an, bei denen wir die Angaben über Methoden durch die Angabe der Tore ersetzen (siehe Abbildung 5.2). Die Typangabe wird in der Deklaration der Instanz hinter dem Tornamen angegeben. Zur Bildung der Typen verwenden wir die *Typkonstruktoren gate* und *returns*. In Abbildung 5.2 wird ein Tor m_1 definiert, dessen Nachrichten eine Zahl als Parameter übertragen können. Daher besitzt dieses Tor den Typ *int gate*. Den Typkonstruktor *gate*

[1] In *Ada* [Shu88, TDT95] werden Tore als *entries* bezeichnet.

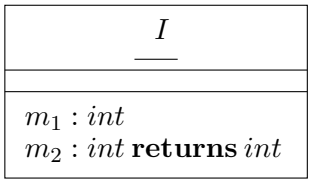


Abbildung 5.2: Deklaration von Instanzen

geben wir aus Gründen der Lesbarkeit in der Definition nicht an. Liefert die Kommunikation über ein Tor einen Wert zurück (s.u.), wird der Rückgabewert nach dem Schlüsselwort **returns** in der Tordeklaration angegeben (siehe m_2 in Abbildung 5.2). Für die Darstellung der mit dem Schlüsselwort **returns** deklarierten Typen verwenden wir den Typkonstruktor *returns*, der den Typ eines Kanals für Anfragen aus dem Typ der anfragenden Nachricht und dem Typ der Antwort bildet[2]. Somit besitzt das Tor m_2 in Abbildung 5.2 den Typ $int\,returns\,int$.

Wir geben die Tordeklarationen oft direkt in der graphischen Darstellung der Instanzen an (siehe die Deklarationen in I_2 in Abbildung 5.3). Die in der Instanzdeklaration angegeben Tornamen sind zu Beginn des Systemablaufs global im System bekannt, so daß alle Instanzen über ihre jeweiligen Tore Nachrichten von anderen Instanzen und der Systemumgebung erhalten können. Zusätzlich besteht die Möglichkeit, Tornamen zur Laufzeit dynamisch zu erzeugen und diese dann durch Kommunikation anderen Instanzen mitzuteilen. Diese Möglichkeit wird in Abschnitt 5.3 erläutert. Die Tore der Umgebung können in einer speziellen Instanzdeklaration namens **env** deklariert werden.

Die Tore einer Instanz müssen unterschiedliche Namen besitzen. Dies gilt auch, falls eine Unterscheidung aufgrund von Anzahl oder Typ der Parameter möglich wäre.

Kommunikation. In diesem Abschnitt beschreiben wir die Interaktionsmöglichkeiten zwischen den Instanzen und ihre graphische Darstellung. Aus Gründen der Einsetzbarkeit von existierenden Entwicklungstools verwenden wir die Syntax aus den Sequenzdiagrammen von UML [BRJ98]. So stellen wir Nachrichten durch Pfeile dar. Ebenso werden unterschiedliche Arten von Nachrichten durch verschiedene Pfeilarten symbolisiert. Weiterhin werden Instanzen in der Syntax der Objekte in UML dargestellt.

Obwohl wir die Syntax für Nachrichten aus UML verwenden, haben die Pfeilarten in unseren Diagrammen, bedingt durch das Systemmodell mit verteilten Instanzen, eine eigene Bedeutung, die sich am Kommunikationsmodell der Prozeßalgebren orientiert. Wir unterscheiden zwei Arten von Nachrichten. Pfeile mit nicht-ausgefüllter Spitze repräsentieren Nachrichten, die von einer Instanz zu ei-

[2]Analog zu Funktionstypen $T_1 \to T_2$ in funktionalen Sprachen [Thi94].

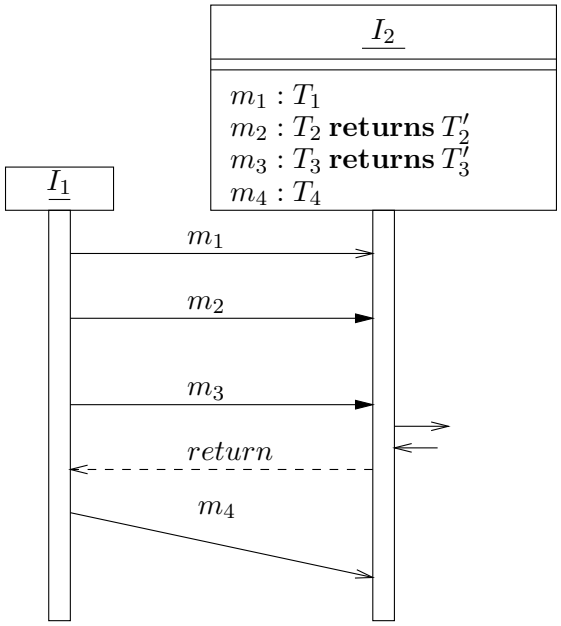


Abbildung 5.3: Pfeilarten in Sequenzdiagrammen.

nem anderen gesendet werden. Die beteiligten Instanzen können direkt nach dem Senden bzw. Empfangen der Nachricht in ihrer Ausführung fortfahren (siehe die Nachricht m_1 in Abbildung 5.3). Pfeile mit ausgefüllten Spitzen stehen für Nachrichten, bei denen der Aufrufer auf eine Antwort des Empfängers warten muß. Diese Art von Nachrichten nennen wir *Anfragen*. Führt der Empfänger zwischen Empfang der Nachricht und Senden der Antwort keine weitere Aktionen durch, ist die zusätzliche graphische Darstellung der Rückantwort nicht notwendig. In Abbildung 5.3 repräsentiert der Pfeil für m_2 sowohl die eigentliche Nachricht als auch die Antwort. Führt der Empfänger Aktionen vor dem Senden der Antwort aus, kann die Antwort als gestrichelter Pfeil dargestellt werden (siehe die Nachricht m_3 in Abbildung 5.3). Wir legen fest, daß die Rückantwort auf eine Anfrage

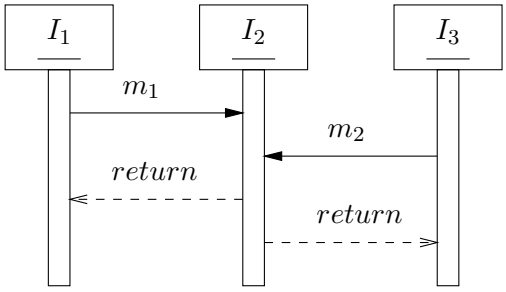


Abbildung 5.4: Nebenläufige Behandlung von Anfragen.

stets den speziellen Tornamen *return* verwenden muß, da für Rückantworten keine eigenen Tore explizit definiert werden. Außerdem wird durch eine einheitliche Beschriftung der Antworten die Lesbarkeit des Diagramms erhöht. Da der Sender einer Anfrage bis zum Eintreffen der Antwort blockiert ist, ist die Zuordnung von Antworten an Anfragen stets eindeutig. Weiterhin erlauben wir auch die gleichzeitige Behandlung von Anfragen verschiedener Instanzen innerhalb einer Instanz (siehe Abbildung 5.4).

Nachrichten können synchron und asynchron ausgetauscht werden. Ein Nachricht ist *synchron*, wenn das Senden und Empfangen der Nachricht von den beiden beteiligten Instanzen gleichzeitig ausgeführt wird. Somit ist die Übertragung einer synchronen Nachricht *ein* atomares Ereignis. Bei *asynchronen* Nachrichten geschieht das Senden und das Empfangen zu verschiedenen Zeitpunkten, d.h. das Senden und das Empfangen der Nachricht sind zwei getrennte Ereignisse; der Sender kann nach dem Senden der Nachricht in seiner Ausführung fortfahren, ohne daß er auf den Empfang der Nachricht durch den Empfänger warten muß. Horizontale Pfeile kennzeichnen synchrone Nachrichten, während diagonale Pfeile asynchrone Nachrichten repräsentieren[3]. Pfeile, die diagonal nach oben gehen, sind verboten. Bei Anfragen muß der Sender stets auf das Eintreffen der Rückantwort warten, unabhängig davon, ob die Anfrage durch eine synchrone oder asynchrone Nachricht realisiert wurde. Die Rückantwort einer Anfrage muß bzgl. synchroner oder asynchroner Kommunikation mit der aufrufenden Nachricht übereinstimmen. Da der Sender sowohl bei einer synchronen als auch bei einer asynchronen Anfrage bis zum Erhalt der Antwort blockiert ist, bezieht sich die Unterscheidung synchron bzw. asynchron nur auf die Übertragung der Nachricht: Bei einer synchronen Nachricht bilden Senden und Empfangen ein atomares Ereignis, während es sich im asynchronen Fall um zwei Ereignisse handelt. Diese Unterscheidung ist besonders im Hinblick auf mögliche Spracherweiterungen sinnvoll, z.B. bei der Spezifikation von Echtzeit-Anforderungen [Dou97]. In der Semantik in Kapitel 6 werden synchrone Nachrichten durch synchrone Kommunikation realisiert. Asynchrone Nachrichten werden durch das Starten spezieller Prozesse nachgebildet, die dann widerum synchron kommunizieren.

Bei synchronen Nachrichten sind der Zeitpunkt des Sendens und des Empfangens identisch. Daher findet z.B. das Senden von m_2 in I_1 in Abbildung 5.3 zum selben Zeitpunkt wie das Empfangen in I_2 statt. Als weitere Zeitbedingung fordern wir nur, daß die Ereignisse auf der Lebenslinie einer Instanz total geordnet sind. Die graphischen Abstände auf den Lebenslinien der Instanzen sind ohne Bedeutung für die Angabe von Zeitintervallen; es läßt sich also z.B. in Abbildung 5.3 nicht bestimmen, ob die zeitliche Distanz zwischen den Sendeereignissen von m_1 und m_2 größer oder kleiner ist als die entsprechende Distanz zwischen m_2 und m_3. Zeitpunkte in verschiedenen Instanzen können nicht miteinander

[3]In UML werden asynchrone Nachrichten durch horizontale Pfeile mit einer halben Pfeilspitze dargestellt [OMG99].

verglichen werden, es sei denn, sie sind durch die Übertragung einer synchronen Nachricht miteinander verbunden. Im Unterschied hierzu besitzt in Message Sequence Charts [ITU96a] jede Instanz eine lokale Zeit. Da in MSCs die Nachrichten nur asynchron übertragen werden, ist hier die Synchronisation zweier Instanzen durch die einfache Übertragung einer synchronen Nachricht nicht möglich; Synchronisation muß hier durch einen Austausch asynchroner Nachrichten simuliert werden.

5.3 Komplexe Diagrammelemente

Interaktionsdiagramme lassen sich auf zwei verschiedene Weisen interpretieren [GF99]. *Exemplarische* Diagramme beschreiben genau eine konkrete Interaktion zwischen den Systemkomponenten. *Generative* Diagramme beschreiben eine Menge von möglichen Interaktionen, die alle korrekte Systemabläufe sind. Die zweite Art von Darstellung wird durch komplexe Sprachelemente wie die Auswahl zwischen Alternativen sowie Schleifen unterstützt.

Interaktionsdiagramme dienen zur Darstellung des Nachrichtenflusses zwischen Systemkomponenten. Daher werden Datentransformationen meist nicht explizit angegeben; Daten treten i. allg. nur als Parameter der Nachrichten oder als Bedingungen in den Beschriftungen der Pfeile auf.

In diesem Abschnitt erweitern wir die in Abschnitt 5.2 vorgestellten Diagramme zur Spezifikation von n-Agenten-Systemen um die Darstellung von Datentransformationen. Weiterhin betrachten wir unsere Diagramme als generative Diagramme, die jeweils eine Menge von möglichen Systemabläufen repräsentieren. Daher integrieren wir Sprachelemente wie Alternativen und Schleifen ähnlich zu den *inline expressions* von MSCs [ITU96a, ITU99], passen diese aber an das Systemmodell der in Abschnitt 5.2 erläuterten Systeme an. Hierbei soll die Syntax von Sequenzdiagrammen so wenig wie möglich verändert werden. Daher ändern wir die Syntax nur in den Bereichen, in denen eine Anpassung der Sequenzdiagramme an die n-Agenten-Systeme notwendig ist.

Wir konzentrieren uns auf eine *funktionale* Beschreibung der Daten von Systemen. Funktionale Sprachen besitzen in der Regel eine wohlfundierte formale Semantik, so daß die Definition einer integrierten formalen Semantik für die Verhaltens- und Datenbeschreibung ermöglicht wird. Um eine möglichst problemorientierte Systembeschreibung zu ermöglichen, abstrahieren wir wie in Kapitel 3 von konkreten Sprachen für die Angabe der Datentransformationen. Stattdessen beschreiben wir generelle Ansätze zur Integration funktionaler Datenbehandlung in unsere Diagramme.

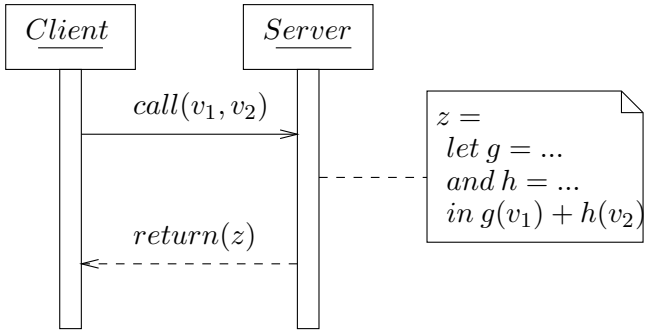


Abbildung 5.5: Beispiel für Annotationen.

5.3.1 Annotationen

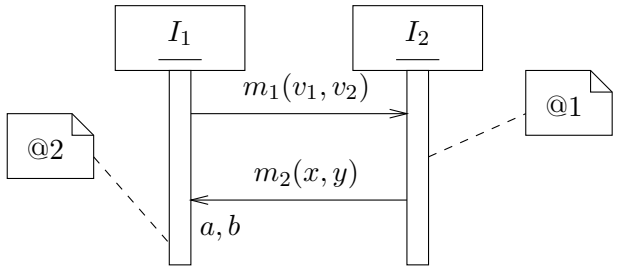
In der Definition von UML ist vorgesehen, Elemente von Systemen, die nicht graphisch spezifiziert werden können, in Form von *Annotationen* (*notes* [OMG99, S. 3-14]) in den Diagrammen anzugeben. Annotationen sind graphische Elemente, die textuelle Informationen beinhalten. Sie werden durch ein Rechteck dargestellt, bei dem die rechte obere Ecke "eingeknickt" ist, und können durch gestrichelte Linien mit einem oder mehreren anderen graphischen Elementen verbunden werden. Typische Inhalte von Annotationen sind Informationen bzgl. des Metamodells von UML, Kommentare, Bedingungen und Methodenrümpfe.

In unserem Ansatz verwenden wir Annotationen zur Darstellung von Datendeklarationen und -berechnungen. In Abbildung 5.5 ist ein Beispiel für ein Sequenzdiagramm mit Datenbeschreibung angegeben. Ein Client sendet mit der Nachricht *call* zwei Werte v_1 und v_2 als Parameter an einen Server[4]. Dieser führt mit diesen Parametern eine Berechnung in einer funktionalen Datensprache durch, bindet das Ergebnis an den Bezeichner z und sendet anschließend das Ergebnis an den Client zurück.

Der Zeitpunkt zur Durchführung einer Datentransformation wird durch den Schnittpunkt zwischen der gestrichelten Linie der Annotation mit der Lebenslinie der entsprechenden Instanz gekennzeichnet. Wir nehmen an, daß Berechnungen auf Daten keine Zeit benötigen und daher durch einen Punkt auf der Lebenslinie darstellbar sind. Terminiert eine funktionale Berechnung nicht, ist die entsprechende Instanz blockiert und kann keine weiteren Aktionen ausführen.

Jeder Bezeichner muß vor seiner Anwendung definiert sein, da andernfalls unreduzierbare offene Terme entstehen können. Werden innerhalb einer Annotation mehrere Bezeichner definiert, entspricht dies der Hintereinanderausführung der Bindungen. Dies geschieht analog zu Termen wie $\{x\leftarrow 1\};\{y\leftarrow 2\};t$ aus $\mathcal{P}$ in Kapitel 3. Ein zuvor definierter Bezeichner kann in den Ausdrücken zur Be-

[4]Wie in den vorhergehenden Kapiteln bezeichnen wir Werte mit $v, v'...$ und Bezeichner mit $x, y, ...$



@1	$x = f(v_1, v_2)$ $y = g(v_1, v_2)$
@2	$z = h(a, b)$

Abbildung 5.6: Diagramm mit Datenverzeichnis.

rechnung der Werte für nachfolgende Bezeichner verwendet werden (vergleiche $\{x \leftarrow 1\}; \{y \leftarrow x + 1\}; t$).

Die Ausdrücke der Datensprache müssen korrekt getypt sein.

Datenverzeichnis. Aus Gründen der Übersichtlichkeit kann es vorteilhaft sein, die Datentransformationen nicht direkt im Sequenzdiagramm anzugeben, sondern in den Annotationen nur Verweise auf Elemente der Datensprache anzugeben. Die Definitionen und Berechnungen werden dann in einem gesonderten *Datenverzeichnis* angegeben. Dieses Datenverzeichnis enthält, vergleichbar mit den *data dictionaries* von DeMarco [DeM79], die Deklarationen von Bezeichnern und Funktionen. Im Gegensatz zu den alphabetisch geordneten *data dictionaries* verweisen wir auf die Einträge mittels Indizes im Diagramm. Wir schreiben den Verweis auf den n-ten Eintrag als @n.

Abbildung 5.6 enthält ein Sequenzdiagramm mit zugehörigem Datenverzeichnis. Im Diagramm treten an zwei Stellen Berechnungen von Daten auf; die entsprechenden Annotationen enthalten nur die Referenzen auf die zugehörigen Einträge im Datenverzeichnis.

Namen von Diagrammen. Um einem Diagramm einen Namen zuzuordnen, kann eine spezielle Annotation verwendet werden, die nach dem Schlüsselwort **diagram** einen Bezeichner als Namen für das Diagramm enthält (siehe Abbildung 5.7). Aus Gründen der Übersichtlichkeit befindet sich diese Annotation meistens am oberen Rand des Diagramms; dies ist aber nicht notwendig, da die Bedeutung der Annotation durch das Schlüsselwort festgelegt ist. Diese Annotation ist nicht durch eine gestrichelte Linie mit anderen graphischen Elementen verbunden, da der Name eines Diagramms eine für das entsprechende Diagramm globale Eigenschaft ist. Aus Gründen der Eindeutigkeit dürfen nicht mehrere Diagramme den gleichen Namen besitzen.

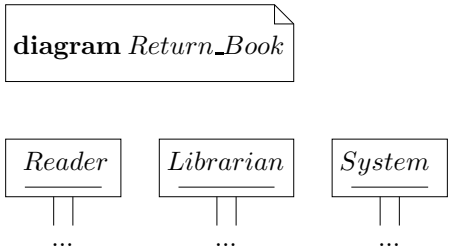


Abbildung 5.7: Diagramm mit Namen.

5.3.2 Gültigkeitsbereiche

Analog zur Verwendung von Bezeichnern in Programmiersprachen ordnen wir Bezeichnern in Sequenzdiagrammen Gültigkeitsbereiche zu. In einem verteilten System ohne globalen Speicher ist die Verwendung globaler Bezeichner nicht sinnvoll. Daher werden Bezeichner stets lokal in einer Instanz definiert und sind dann auch nur in der entsprechenden Instanz bekannt.

Parameter in Nachrichten. Als Auswertungsstrategie für die Parameter in Nachrichten verwenden wir die Strategie der *eager-evaluation* (*call-by-value*): Die Ausdrücke in der Parameterliste werden vor dem Senden der Nachricht zu Werten reduziert und diese dann an den Empfänger der Nachricht übertragen. In den Ausdrücken der Parameterliste dürfen nur die in der sendenden Instanz bekannten Bezeichner vorkommen.

Bindung empfangener Werte. Zur Angabe der Bezeichner, an die Parameter einer empfangenen Nachricht gebunden werden sollen, kann eine Annotation an die Spitze des die Nachricht repräsentierenden Pfeils angefügt werden. Diese Annotation enthält nach dem Schlüsselwort **bind** eine Liste der Bezeichner, an die die Werte der Parameter gebunden werden sollen. Dabei wird der i-te Parameter an den i-ten Bezeichner der Liste gebunden. Ein Beispiel für eine solche Annotation ist in Abbildung 5.8 angegeben. Hier werden die von m_1 übertragenen Werte in I_2 an die Bezeichner y_1 und y_2 gebunden, die dann in weiteren Berechnungen verwendet werden.

Da die Bindung an Bezeichner häufig in Diagrammen vorkommt, erlauben wir eine alternative Syntax, bei der die Bezeichner direkt ohne Annotation an die Pfeilspitze geschrieben werden. Diese Variante verbessert die Übersichtlichkeit der Diagramme, da zum einen die Anzahl der graphischen Elemente vermindert wird, und zum anderen die Information über die Bezeichner sich direkt an der zugehörigen Nachricht befindet. Allerdings geht die Kompatibilität mit vielen Entwicklungswerkzeugen verloren. In Abbildung 5.8 werden die mit m_2 übertragenen Werte in I_1 an die Bezeichner b_1 und b_2 gebunden.

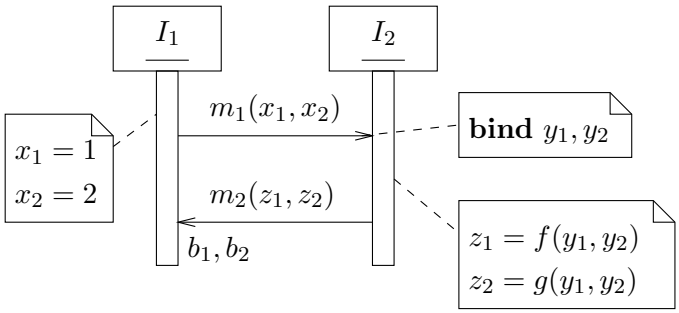


Abbildung 5.8: Bindung von gelesenen Werten an Bezeichner.

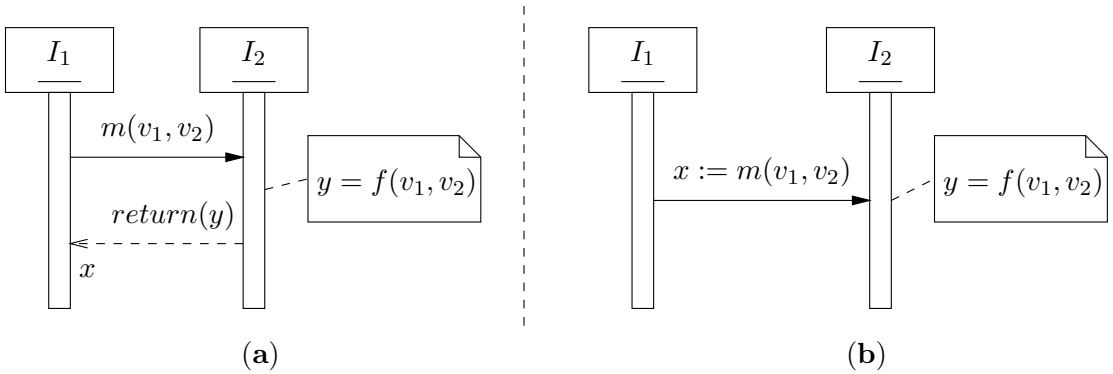


Abbildung 5.9: Abkürzende Notation für Anfragen.

Anfragen. In Systemen mit parallelen Komponenten treten häufig Anfragen auf, bei denen die aufrufende Komponente nach dem Senden der Anfrage auf den Erhalt eines Ergebnisses wartet und die die Anfrage bearbeitende Komponente vor dem Senden der Antwort nur Datenoperationen ausführt. Aus Gründen der Übersichtlichkeit definieren wir eine abkürzende Notation für solche Anfragen. In Abbildung 5.9(a) ist eine Anfrage dargestellt, bei der die sendende Komponente I_1 auf die Rückantwort der Instanz I_2 warten muß. Die Rückantwort wird, wie bereits in Abbildung 5.3 gezeigt, durch einen Pfeil mit gestrichelter Linie dargestellt. Zwischen Empfang der Anfrage und Senden der Antwort führt I_2 außer der Berechnung von Daten keine Aktionen aus. Der in der Antwort gesendete Wert wird in I_1 an den Bezeichner x gebunden. Sind diese Bedingungen erfüllt, kann die Anfrage, wie in Abbildung 5.9(b) angegeben, durch einen einfachen Pfeil ersetzt werden. Die Bindung des Ergebnisses an den Bezeichner x wird in Form einer Zuweisung in der Beschriftung des Nachrichtenpfeils dargestellt. In der Rückantwort werden die Werte der Bezeichner übermittelt, die in der Annotation definiert werden; in Abbildung 5.9 also der Wert von y. Werden mehrere Bezeichner definiert,

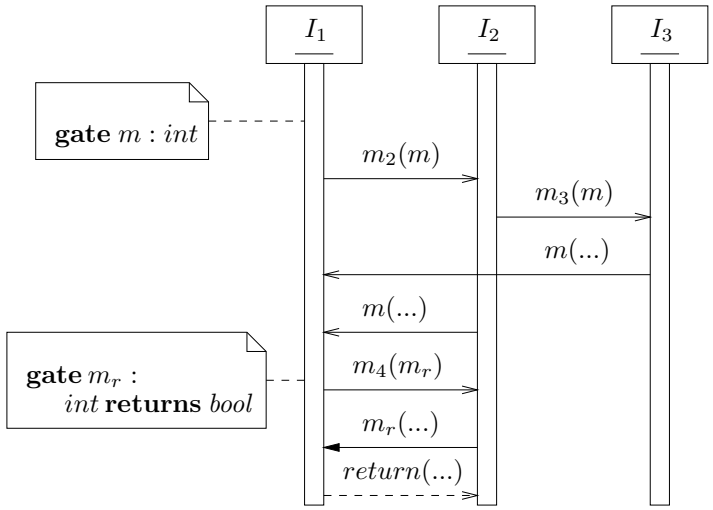


Abbildung 5.10: Dynamische Erzeugung von Tornamen.

werden diese Bezeichner als Parametervektor in der Reihenfolge ihrer Definition in der Antwortnachricht übertragen.

Dynamische Erzeugung von Toren. Wie bereits in Abschnitt 5.2 angeführt, erlauben wir die dynamische Erzeugung von Toren zur Laufzeit eines Systems.

Die Erzeugung eines neuen Tores wird in einer Annotation dargestellt, die das Schlüsselwort **gate**, gefolgt von einer Liste aus Tornamen und zugehörigen Typen, enthält.

In Abbildung 5.10 wird in I_1 dynamisch ein neues Tor mit dem Namen m erzeugt. Mit der Nachricht m_2 wird m der Instanz I_2 bekanntgemacht. I_2 reicht dann das Tor an I_3 weiter. Die Bindung von empfangenen Tornamen an Bezeichner muß im Empfänger nicht explizit angegeben werden, da die Tornamen implizit an einen gleichlautenden Bezeichner gebunden werden. Nach dem Empfang von m senden sowohl I_2 als auch I_3 Nachrichten über das Tor m an I_1. Die Tore m_2 und m_3 in Abbildung 5.10 sind daher vom Typ *int gate gate*.

Weiterhin wird in in Abbildung 5.10 ein neues Tor m_r erzeugt, über das Anfragen an I_1 gesendet werden können. Dieses Tor wird an I_2 über das Tor m_4 übermittelt. Daher besitzt dieses Tor den Typ (*int returns bool*) *gate*.

Während das Recht, Nachrichten über ein Tor zu senden, durch die Weitergabe des Tores an andere Instanzen übertragen werden kann, ist das Empfangen von Informationen über ein Tor nur der Instanz möglich, in der das Tor erzeugt wurde. Daher können die Instanzen I_2 und I_3 in Abbildung 5.10 keine Informationen über die Tore m und m_r empfangen.

Die Gültigkeit eines neuen Tornamens beschränkt sich auf das Diagramm, in dem er erzeugt wurde. Soll ein neuer Name in einem nachfolgenden Diagramm

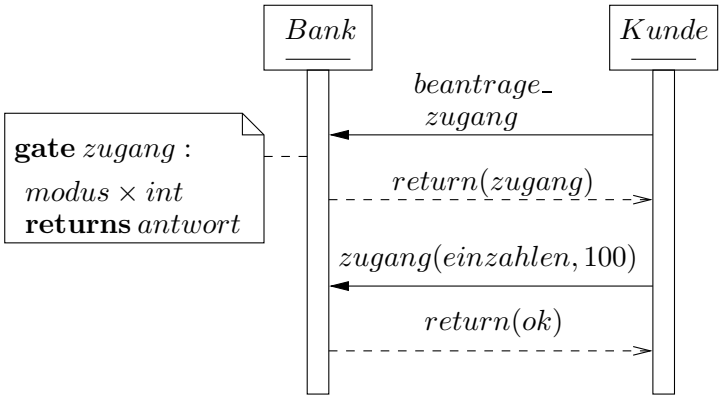


Abbildung 5.11: Beispiel für sichere Interaktion durch dynamische Kanäle.

verwendet werden, muß er als Parameter an das nachfolgende Diagramm übergeben werden (siehe Abschnitt 5.3.6).

Beispiel 5.2 *In Abbildung 5.11 ist eine Beispiel angegeben, das die dynamische Erzeugung von Toren zur Modellierung von Zugriffsrechten verwendet. Ein Kunde beantragt bei einer Bank einen Zugang für* electronic banking*. Nach dem Erhalt der Anfrage des Kunden generiert die Bank eines neues Tor, über das der Kunde dann seine Aufträge senden kann. Nachrichten über das Zugangstor enthalten als Parameter einen Wert des Typs modus, der die Art des Zugangs regelt (z.B. einzahlen oder abheben), sowie eine Zahl für Geldbeträge. Als Antwort sendet die Bank dann in der Antwortnachricht einen Wert des Typs antwort (z.B. ok oder fehler). Das neue Zugangstor wird in der Antwort der Anfrage beantrage_zugang übermittelt, so daß nur dieser spezielle Kunde Kenntnis von diesem Tor hat. Daher kann auch nur dieser Kunde Aufträge über dieses Tor an die Bank senden. Erst wenn der Kunde das Tor anderen Kunden bekanntgibt, können auch diese Aufträge über das Zugangstor senden.*

Es zeigt sich, daß sich die Handhabung von Zugriffsrechten durch dynamische Tore besonders effizient modellieren läßt, da die dynamischen Tore nur nach expliziter Weitergabe von anderen Instanzen verwendet werden können.

5.3.3 Darstellung von Alternativen

Bevor wir auf die Realisierung der Auswahloperation der n-Agenten-Diagramme eingehen, möchten wir zunächst zur Motivation die entsprechenden Operationen aus den Interaktionsdiagrammen von UML und den Message Sequence Charts diskutieren.

Die Interaktionsdiagramme in UML [BRJ98] ermöglichen die Beschreibung alternativen Verhaltens durch die Angabe bedingter Nachrichten. Hierbei wird

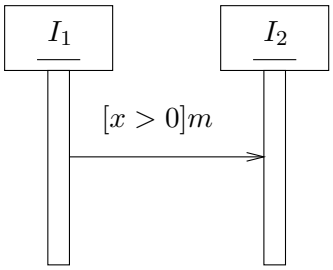


Abbildung 5.12: Sequenzdiagramm mit Bedingung.

das Versenden einer Nachricht von einer Bedingung abhängig gemacht (siehe z.B. die Nachricht *notify* in Abbildung 5.1).

Die Verwendung bedingter Nachrichten ist in Sequenzdiagrammen, die zur Beschreibung nebenläufiger Teilsysteme verwendet werden, problematisch [GGW98, GGW99]. Um bedingte Nachrichten korrekt zu behandeln, muß die Bedingung sowohl im Sender als auch im Empfänger der Nachricht berücksichtigt werden. Wir erläutern dies am Beispiel des Sequenzdiagramms in Abbildung 5.12, in dem eine Instanz I_1 eine bedingte Nachricht m an die Instanz I_2 sendet.

- I_2 ist ein aktiver Prozeß, daher muß I_2 mit seiner Ausführung auch fortfahren können, wenn die Nachricht m nicht gesendet wird. Daher muß I_2 wissen, ob I_1 die Nachricht senden wird, damit seine weitere Ausführung nicht durch ein vergebliches Warten auf m blockiert wird[5].

- Die Entscheidung für das Senden von m wird von der Instanz I_1 getroffen. Daher kann die Bedingung, die über das Senden entscheidet, alle Bezeichner verwenden, die zum Zeitpunkt der Entscheidung in I_1 bekannt sind. Da I_2 ebenfalls entscheiden können muß, ob m gesendet wird, muß die Auswertung der Bedingung auch in I_2 möglich sein. Dies impliziert eine Art von globalem Speicher, der ermöglicht, daß I_2 auf die Werte aller, also auch der lokalen, Bezeichner von I_1 zugreifen kann. Diese Annahme ist in einem verteilten System nicht realistisch; außerdem wird hierdurch das Prinzip der Lokalität verletzt.

- Werden in I_2 die von m übertragenen Werte an Bezeichner gebunden, die in späteren Aktionen verwendet werden, erfolgt diese Bindung bei Nichtsenden von m nicht. Dies führt dazu, daß Bezeichner eventuell nicht mit Werten belegt werden, so daß offene Terme entstehen, die nicht reduziert werden können. Eine mögliche Lösung dieses Problems besteht in der Angabe von *default*-Werten, die bei Nichtsenden von m an die Bezeichner gebunden

[5] In einer Implementierung könnte die Blockierung durch eine Zeitbeschränkung des Wartens (*timeout*) verhindert werden.

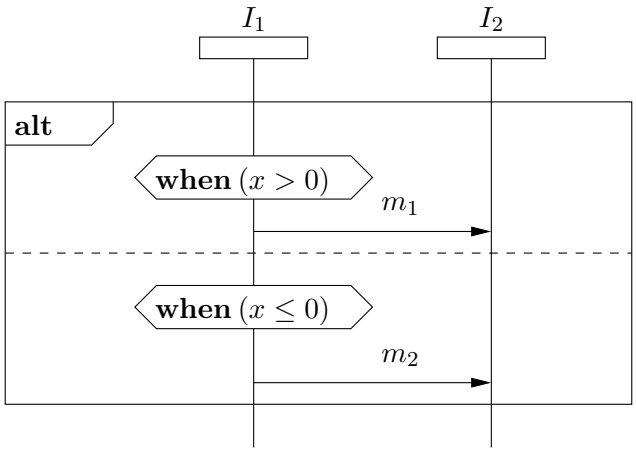


Abbildung 5.13: Message Sequence Chart mit Alternative.

werden. Diese Möglichkeit wird aber kaum in existierenden parallelen Programmiersprachen unterstützt, so daß eine Umsetzung der Diagramme in Programme erschwert würde.

In den Message Sequence Charts [ITU96a, ITU99, Ren99] gibt es das Konzept der *inline expressions* zur Darstellung komplexen Verhaltens. Sie dienen zur Komposition von MSCs innerhalb anderer MSCs. Die zu verknüpfenden MSCs werden innerhalb eines Rahmens übereinander gezeichnet und durch gestrichelte Linien getrennt.In der oberen linken Ecke des Rahmens wird die Art der Komposition durch ein Schlüsselwort angegeben. Mögliche Schlüsselwörter sind z.B. **alt** für alternative Komposition oder **par** für Parallelkomposition [Ren99].

In Abbildung 5.13 ist eine alternative Komposition dargestellt. In Abhängigkeit des Wertes des Bezeichners x wird entweder die Nachricht m_1 oder die Nachricht m_2 an die Instanz I_2 gesendet. Im bisherigen Standard MSC '96 [ITU96a, Rep99] besitzen Bedingungen (*conditions*) nur den Status von Kommentaren, so daß die Angabe der Bedingungen keine semantische Bedeutung besitzt. Daher ist die Auswahl zwischen den beiden Alternativen, die informell von x abhängt, in der formalen Semantik nichtdeterministisch. Im Standard MSC 2000 ist Datenbehandlung integriert, so daß die Auswahl in Abhängigkeit von x getroffen werden kann; allerdings können auch hier immer noch nichtdeterministische Auswahloperationen angegeben werden, da die Spezifikation von Bedingungen nicht zwingend ist. Wir gehen im Rahmen der Konkatenation von Diagrammen in Abschnitt 5.5.2 auf die Problematik der Mehrdeutigkeit der Komposition von Diagrammen ein, die auch für die alternative Komposition gelten.

Hinzu kommt, daß die graphische Darstellung der Alternativen vertikal übereinander nicht dem Zeitfluß in den Diagrammen entspricht.

Aus den genannten Gründen verwenden wir weder die Notation der bedingten

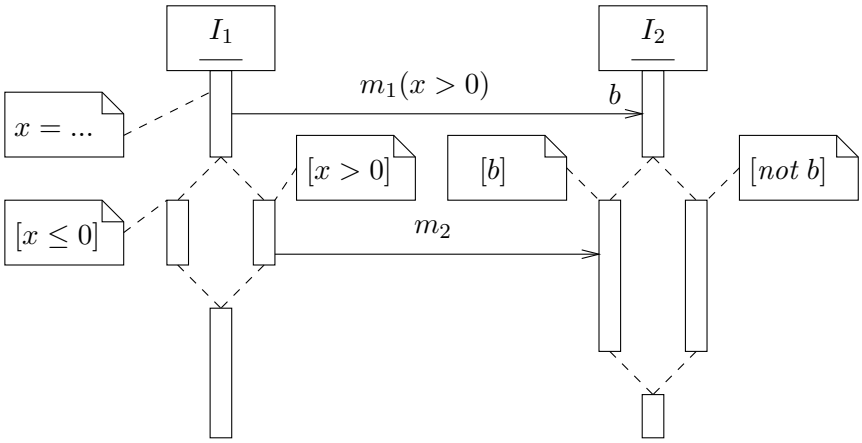


Abbildung 5.14: Sequenzdiagramm mit Auswahloperatoren.

Nachrichten aus UML [BRJ98, OMG99] noch die *inline expressions* aus den MSCs [ITU96a, ITU99, Ren99] für die Spezifikation von n-Agenten-Systemen. Daher definieren wir eine eigene Notation für alternatives Verhalten, bei der die Auswahl der Alternative lokal innerhalb der entsprechenden Instanz getroffen wird. Hierbei ist zu beachten, daß durch die Verwendung unserer Auswahloperation die syntaktische Kompatibilität zu existierenden Entwurfswerkzeugen verlorengeht.

In den n-Agenten-Diagrammen findet die Auswahl zwischen Alternativen jeweils nur innerhalb einer Instanz statt. Die Auswahl wird nur in Abhängigkeit von den lokalen Daten der Instanz getroffen, so daß eine Auswahl nicht direkt durch Interaktion von anderen Instanzen beeinflußt werden kann. Syntaktisch wird eine Auswahlaktion durch eine Aufteilung der Lebenslinie der betreffenden Instanz dargestellt. Die Bedingungen der Alternativen werden in Annotationen in der gewohnten UML-Notation für Bedingungen mit eckigen Klammern angegeben. Können Bedingungen mehrerer Alternativen zu *true* ausgewertet werden, wird nichtdeterministisch eine der Alternativen ausgewählt. Aus Gründen der Übersichtlichkeit kann die Bedingung auch direkt ohne den Annotations-Rahmen neben den Beginn der Lebenslinie der Alternative geschrieben werden.

In Abbildung 5.14 wird in beiden Instanzen eine Auswahl zwischen zwei Alternativen getroffen. Zuerst wird in I_1 ein Bezeichner x definiert, an den das Ergebnis einer Berechnung gebunden wird. Das Resultat der Bedingung $x > 0$ wird dann mit der Nachricht m_1 an I_2 gesendet, wo es an den Bezeichner b gebunden wird. Gilt $x > 0$, wird in I_1 die zweite Alternative eingegangen, die eine Nachricht m_2 an die Instanz I_2 sendet. Entsprechend entscheidet sich I_2 in Abhängigkeit von dem Wahrheitswert in b für den Empfang der Nachricht.

Der Gültigkeitsbereich von Bezeichnern, die innerhalb der Lebenslinie einer Alternative definiert werden, erstreckt sich nur auf diese Alternative. Würde sich der Gültigkeitsbereich eines in einer Alternative definiert Bezeichners über die

Alternative hinaus erstrecken, müßte gewährleistet sein, daß in allen Alternativen dieselbe Menge von Bezeichnern definiert wird, da sonst durch die Auswahl offene Terme entstehen können, die nicht reduziert werden können.

Ebenso muß die Rückantwort auf eine Anfrage (siehe Abbildung 5.3), die in einer Alternative empfangen wurde, innerhalb derselben Alternative an den Aufrufer gesendet werden, damit eine eindeutige Zuordnung von Nachricht und Rückantwort möglich ist.

Unsere Version der Modellierung von Alternativen, die nur von lokalen Entscheidungen der Instanzen abhängig sind, setzt natürlich weiterhin voraus, daß der vom Benutzer angegebene Informationsfluß sinnvoll ist. So würde in Abbildung 5.14 die Vertauschung der Bedingungen in I_2 zur einer Blockierung (*deadlock*) führen, da sich die Bedingungen für das Senden von m_2 in I_1 und für das Empfangen in I_2 ausschließen würden. Unsere Modellierung von Alternativen vermeidet aber die oben skizzierten Probleme bei der semantischen Interpretation der bedingten Nachrichten von UML und der *inline expressions* der MSCs, so daß die Umsetzung in Programmcode vereinfacht wird.

Alternative Eingabeaktionen. Wir führen als weiteres syntaktisches Konstrukt die Auswahl zwischen mehreren Möglichkeiten zum Empfang einer Nachricht ein. In Abbildung 5.15(a) trifft die Instanz I_1 eine Auswahl zwischen zwei alternativen Ausführungen. In jeder Alternative sendet I_1 die Nachricht m an die Instanz I_2, wobei unterschiedliche Werte übertragen werden. Das alternative Empfangen von $m(v_1)$ oder $m(v_2)$ stellen wir in I_2 durch eine gemeinsamen Endpunkt der beiden Pfeile dar. Je nach Wahl der Alternative in I_1 erhält somit der Bezeichner x in I_3 den Wert v_1 bzw. v_2. Alternative Möglichkeiten einer Nachricht können von verschiedenen Instanzen gesendet werden; in Abbildung 5.15(b) wird eine Interaktion spezifiziert, bei der I_3 die Nachricht m entweder von I_1 oder von I_2 empfängt. Zuvor hat I_3 den anderen beiden Instanzen den gewünschten Sender durch die Übertragung eines booleschen Wertes mitgeteilt.

Folgende Anforderungen müssen erfüllt sein, um die Auswahl zwischen Eingabeaktionen anwenden zu können:

- Die beteiligten Nachrichten müssen dasselbe Tor adressieren.
- Alle Nachrichten müssen einheitlich synchron bzw. asynchron übertragen werden.
- Es darf sich nicht um eine Anfrage handeln.
- Es muß vom Benutzer durch die Angabe des Datenflusses gewährleistet sein, daß nur eine der möglichen Nachrichten gesendet wird. Dies kann z.B. durch eine explizite Information der beteiligten Instanzen wie in Abbildung 5.15(b) realisiert werden.

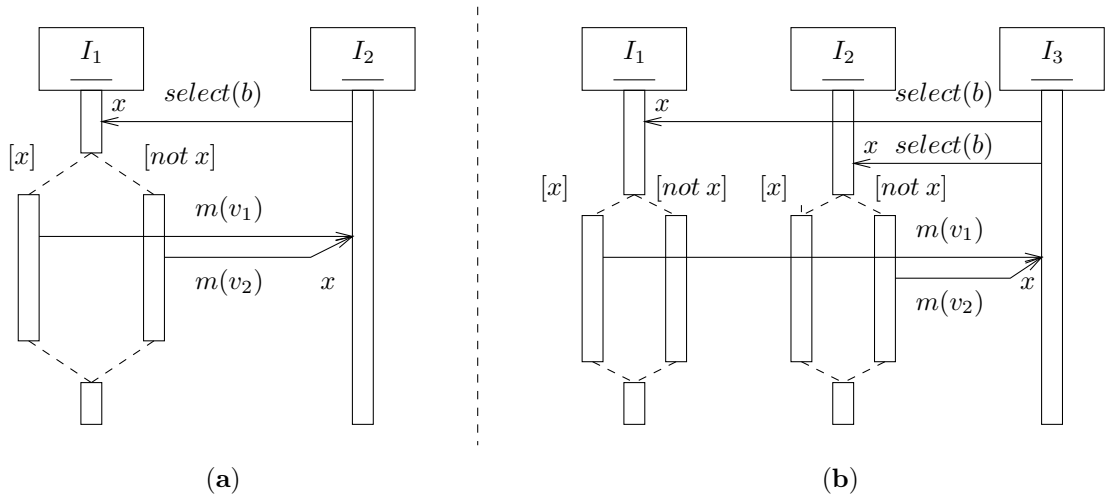


Abbildung 5.15: Diagramm mit alternativen Eingabeaktionen.

Die alternative Eingabe erlaubt die Interaktion mit Instanzen, die intern eine Auswahl zwischen mehreren Alternativen durchführen, ohne daß die interagierende Instanz ebenfalls eine Auswahl treffen muß. So erhält in Abbildung 5.15(a) die Instanz I_2 in jedem Fall einen Wert von I_1 über das Tor m_1, unabhängig von der in I_1 getroffenen Auswahl. Daher muß I_2 nicht über die Auswahl zwischen Alternativen in I_1 informiert sein. Ohne die alternative Eingabe müsste I_2 ebenfalls eine Auswahl treffen, um das Empfangen alternativer Werte von I_1 modellieren zu können.

5.3.4 Schleifen

In UML wird die Verwendung von Schleifen in Interaktionsdiagrammen nicht explizit behandelt. In [OMG99, S. 3-101] wird nur angegeben, daß die wiederholte Ausführung von Teilen einer Interaktion durch das Einrahmen der entsprechenden Nachrichten mit einem Kasten dargestellt werden soll. Am Boden dieses Kastens soll dann die Schleifenbedingung angegeben werden.

Bei der Verwendung dieses Schleifenkonstrukts in Diagrammen, die verteilte Systeme darstellen, treten ähnliche Problem wie bei Alternativen auf (siehe Abschnitt 5.3.3). Da die Schleifenbedingung global für alle an der Schleife beteiligten Instanzen definiert wird, ist nicht klar, wie die einzelnen Instanzen die Gültigkeit der Bedingung ohne die Verwendung eines gemeinsamen globalen Speichers überprüfen können.

In MSCs werden Schleifen wie die Auswhl von Alternativen als *inline expressions* realisiert [ITU96a, Ren99, ITU99]. Die Schleifenbedingung wird hinter dem Schlüsselwort **loop** in der linken oberen Ecke des Rahmens angegeben, der das

zu iterierende Verhalten eingrenzt. Da die Bedingung nicht explizit einer Instanz zugeordnet wird, ist nicht ersichtlich, welche Bezeichner in der Bedingung auftreten dürfen. Weiterhin wird nicht definiert, wie die Schleifenkontrolle in einem verteilten System mit nur lokalem Speicher realisiert wird. Insbesondere wird in dem Schleifenkonstrukt nicht explizit festgelegt, welche Systemkomponente die Entscheidung über die Fortführung der Iteration trifft.

Wir führen für die n-Agenten-Diagramme ein eigenes Schleifenkonzept ein. Wir stellen im folgenden zwei Arten von Schleifen vor, die sich durch die Realisierung der Iterationskontrolle unterscheiden. Beide Schleifenkonzepte sind speziell auf das Systemmodell der verteilten Instanzen hin entworfen. Vor der Vorstellung der Schleifenarten diskutieren wir die Iteration von Verhalten in Verbindung mit funktionaler Programmierung.

Schleifen und Bindung an Bezeichner. Iteration ist ein Konstrukt aus imperativen Programmiersprachen, in denen der Zustand eines Programms durch die aktuelle Belegung seiner Variablen definiert ist. Die Ausführung eines Schleifenrumpfs führt i. allg. zu einer Veränderung des Programmzustands, da Variablen mit neuen Werten belegt werden. Ebenso ist die Entscheidung über die Durchführung weiterer Iterationsschritte von dem jeweiligen Zustand abhängig.

In unserem Ansatz verwenden wir hingegen keine imperativen Datensprachen, sondern funktionale Sprachen. Funktionale Sprachen gehören zu den deklarativen Sprachen und kennen daher weder veränderliche Variablen noch einen expliziten Zustandsbegriff. Somit müssen wir die Schleifenkonstrukte der n-Agenten-Diagramme an die deklarative Behandlung von Daten anpassen.

Wir realisieren die Belegung von Variablen in imperativen Sprachen in unseren Schleifen durch die sukzessive Einführung neuer Bezeichner. Werden in einem Schleifenrumpf Bezeichner definiert, so wird in jedem Iterationsschritt ein neuer Bezeichner mit dem gleichen Namen erzeugt, der den bisherigen Bezeichner gleichen Namens überdeckt. Dies geschieht analog zu einem Term mit Definitionsoperatoren $\{x \leftarrow 1\}; \{x \leftarrow x + 2\}; c!x$ des Kalküls $\mathcal{P}$ aus Kapitel 3 (vergleiche Seite 125). Hier wird zuerst ein Bezeichner x definiert, an den der Wert 1 gebunden wird. Anschließend wird ein weiterer Bezeichner x deklariert, dessen Wert unter Verwendung des vorherigen x zu 3 berechnet wird. Der neue Bezeichner x überdeckt nun die Gültigkeit des alten Bezeichners, so daß in der Ausgabeaktion der Wert des neuen Bezeichners übertragen wird. Auf diese Weise entspricht die Iteration des Schleifenrumpfs der mehrfachen sequentiellen Komposition des Schleifenrumpfs.

In jedem Iterationsschritt einer Schleife sind die Bezeichner des jeweils vorhergehenden Iterationsschritts gültig, so lange sie nicht durch neue Definitionen überdeckt werden. Ebenso greift die Schleifenbedingung auf die Bezeichner des jeweils letzten Iterationschritts zu. In Abbildung 5.17 ist eine Schleife angegeben, in der in jedem Iterationsschritt ein neuer Bezeichner max definiert wird.

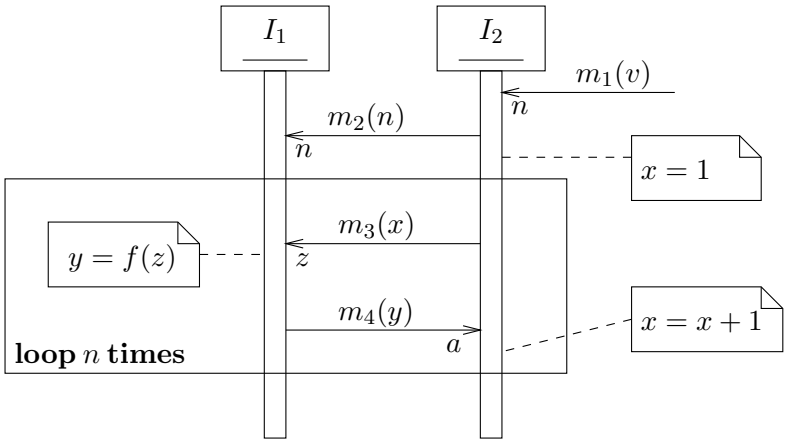


Abbildung 5.16: *loop*-Schleifen.

Dieser Bezeichner überdeckt die anderen Bezeichner gleichens Namens aus den vorangehenden Iterationsschritten. Nach dem letzten Iterationsschritt der Schleife sind die Bezeichner dieses letzten Schritts gültig. So gilt in Abbildung 5.17 nach der Schleife der Bezeichner max, der im letzten Iterationsschritt definiert wurde. Dieser Wert wird an die Umgebung gesendet.

Loop–Schleife. Der erste Schleifentyp ist eine Schleife, bei der die Anzahl der Schleifendurchläufe vor Beginn der ersten Iteration feststeht. Zudem muß jede Instanz vor Beginn der Schleife über die Anzahl der Schleifendurchläufe informiert sein. Wir bezeichnen diese Schleifenform als *loop*-Schleife. Zur korrekten Durchführung der Schleife ist es notwendig, daß der Wert für die Anzahl der Schleifendurchläufe in allen beteiligten Instanzen an denselben Bezeichner gebunden ist. Die Einhaltung dieser semantischen Bedingung obliegt dem Benutzer, der ihre Gültigkeit durch entsprechende Interaktion zwischen den beteiligten Instanzen realisieren muß. Die Schleife wird durch **loop** *id* **times** gekennzeichnet, wobei *id* ein Bezeichner ist, an den in allen beteiligten Instanzen der Wert für die Anzahl der Schleifendurchläufe gebunden ist. Ist zu Beginn der Schleife der Wert von *id* gleich oder kleiner als 0, wird keine Iteration durchgeführt.

In Abbildung 5.16 ist eine Schleife mit festgelegter Iterationsanzahl angegeben. Die Instanz I_2 erhält aus der Umgebung die Anzahl der Iterationen und bindet diesen Wert an den Bezeichner n. Anschließend wird der Wert von n nach I_1 übertragen und dort ebenfalls an einen Bezeichner n gebunden. In I_2 wird anschließend der Bezeichner x mit dem Wert 1 initialisiert. Der Schleifenrumpf wird nun n-mal durchlaufen. In jedem Iterationsschritt sendet I_2 den Wert des jeweiligen Bezeichners x an I_1 und erhält von dort den Wert einer Berechnung zurück. Anschließend wird *neuer* Bezeichner x eingeführt, an den der um 1 inkrementierte Wert des vorherigen Bezeichners x gebunden wird.

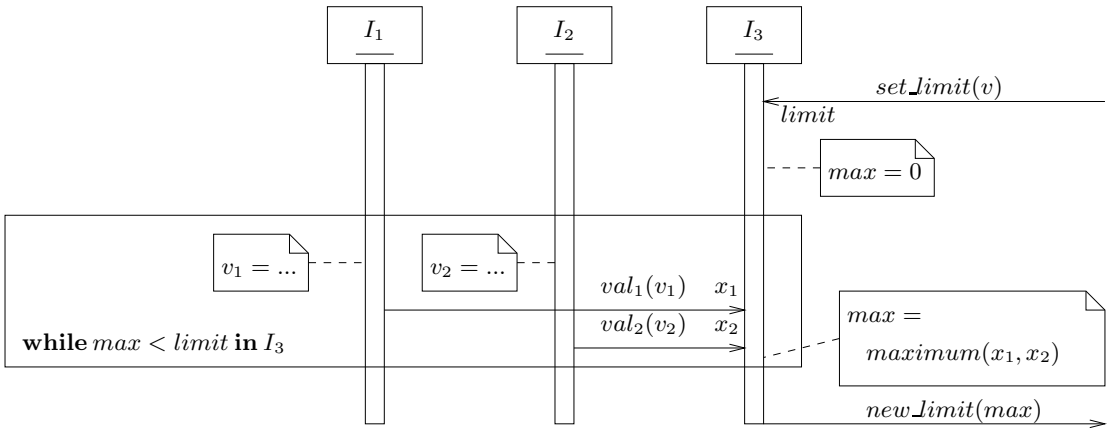


Abbildung 5.17: *while*-Schleife.

While–Schleife. Der zweite Schleifentyp ist eine *while*-Schleife, bei der eine ausgewählte Instanz als Kontrollinstanz fungiert. Diese Instanz steuert den Schleifenablauf, indem sie für jede Iteration lokal die Schleifenbedingung auswertet. Das Resultat der Auswertung wird dann durch implizite, synchrone Nachrichten, die im Diagramm nicht explizit dargestellt werden, an die anderen Instanzen übertragen. Die hierbei verwendeten Tore werden ebenfalls als implizit deklariert angesehen und besitzen den Typ *bool*. Die Instanzen führen dann in Abhängigkeit von der empfangenen Nachricht eine weitere Iteration durch bzw. beenden die Schleife. Analog zu Programmiersprachen [JW85, TDT95] ist die *while*-Schleife eine abweisende Schleife, da die Gültigkeit der Bedingung vor Beginn eines Iterationsschritts überprüft wird. Daher kann es auch vorkommen, daß eine Schleife gar nicht durchlaufen wird. Die Umgebung **env** kann nicht als Kontrollinstanz verwendet werden.

In Abbildung 5.17 ist ein Diagramm mit einer *while*-Schleife angegeben. Die Instanz I_3 fungiert als Kontrollinstanz der Schleife; die Schleife wird so lange durchlaufen, bis der an den Bezeichner max gebundene Wert größer ist der Wert von $limit$, den die Instanz aus der Umgebung erhalten hat. Zu Beginn jedes Schleifendurchlaufs signalisiert I_3 den anderen Instanzen I_1 und I_2, ob eine weitere Iteration erfolgen soll. Das Senden und Empfangen dieser Nachrichten wird im Diagramm nicht dargestellt, sondern geschieht implizit synchron vor der ersten dargestellten Nachricht im Schleifenrumpf. Im Beispiel in Abbildung 5.17 berechnen die Instanzen I_1 und I_2 lokale Werte v_1 bzw. v_2, aus denen die Kontrollinstanz I_3 das Maximum bestimmt. Ist dieses Maximum kleiner als das gesetzte Limit, wird die Schleife erneut ausgeführt. Andernfalls terminiert die Schleife und das Maximum wird als neues Limit der Umgebung mitgeteilt.

Verschachtelte Schleifen. Schleifen können beliebig ineinander verschachtelt werden. Auch bei der Realisierung verschachtelter *while*-Schleifen ist pro Instanz nur ein implizites Tor für die Verwaltung der Schleife notwendig, da die Nachrichten zur Schleifensteuerung synchron übertragen werden. Somit ist sichergestellt, daß die Steuernachrichten stets in der richtigen Reihenfolge in den Instanzen eintreffen und daher eine Zuordnung der Nachrichten zu der entsprechend aktuellen Schleifentiefe möglich ist.

Schleifen dürfen nicht innerhalb der Alternativen eines Auswahloperators auftreten, da sonst gewährleistet sein muß, daß eventuelle Kommunikationspartner ebenfalls die Iteration ausführen können.

In der formalen Semantik der n-Agenten-Diagramme in Kapitel 6 beschränken wir die Schachtelungstiefe von *loop*-Schleifen auf 1. Der Grund hierfür ist eine vereinfachte Angabe der Übersetzungsfunktionen. Es werden Hinweise gegeben, wie eine beliebige Verschachtelung von *loop*-Schleifen in der Semantik realisiert werden kann.

5.3.5 Beispiel: Gefangenendilemma

Als Beispiel für ein Sequenzdiagramm mit Daten spezifizieren wir das aus der Spieltheorie bekannte *Gefangenendilemma* (z.B. in [Hof88, S. 781 ff]). An diesem Spiel sind zwei Spieler beteiligt. Zu Beginn des Spiels wird eine feste Anzahl von Spielrunden festgelegt. In jeder Spielrunde können die Spieler aus zwei möglichen Alternativen wählen: Ein Spieler kann mit seinem Gegner kooperieren oder ihn betrügen. Die Bewertung einer Spielrunde wird durch einen Vergleich der Züge der beiden Spieler vorgenommen:

- Entscheiden sich beide Spiele für Kooperation, erhalten beide Spieler drei Punkte.
- Will einer der beiden Spieler kooperieren und der andere Spieler betrügt ihn, erhält der Betrüger fünf Punkte und der Betrogene keinen Punkt.
- Entscheiden sich beide Spieler für den Betrug, erhält jeder von ihnen einen Punkt.

Das Ziel des Spieles besteht darin, durch eine möglichst optimale Strategie von Kooperation und Betrug die größte Anzahl von Punkten zu erhalten, um das gesamte Spiel zu gewinnen. Das Spiel wird durch einen Schiedsrichter koordiniert, der die Züge der Spieler entgegennimmt, die Punktekonten aktualisiert und den Spielern die gegnerischen Züge mitteilt.

In Abbildung 5.18 ist der Ablauf eines Spiels dargestellt. Als Datensprache verwenden wir die in Kapitel 4 vorgestellte funktionale Sprache. Das Datenverzeichnis zu Abbildung 5.18 ist in Abbildung 5.19 angegeben.

Die Menge der Instanzen beinhaltet die beiden Spieler und den Schiedsrichter. Der Schiedsrichter besitzt die Tore $wahl_A$ und $wahl_B$, über die er die Züge der

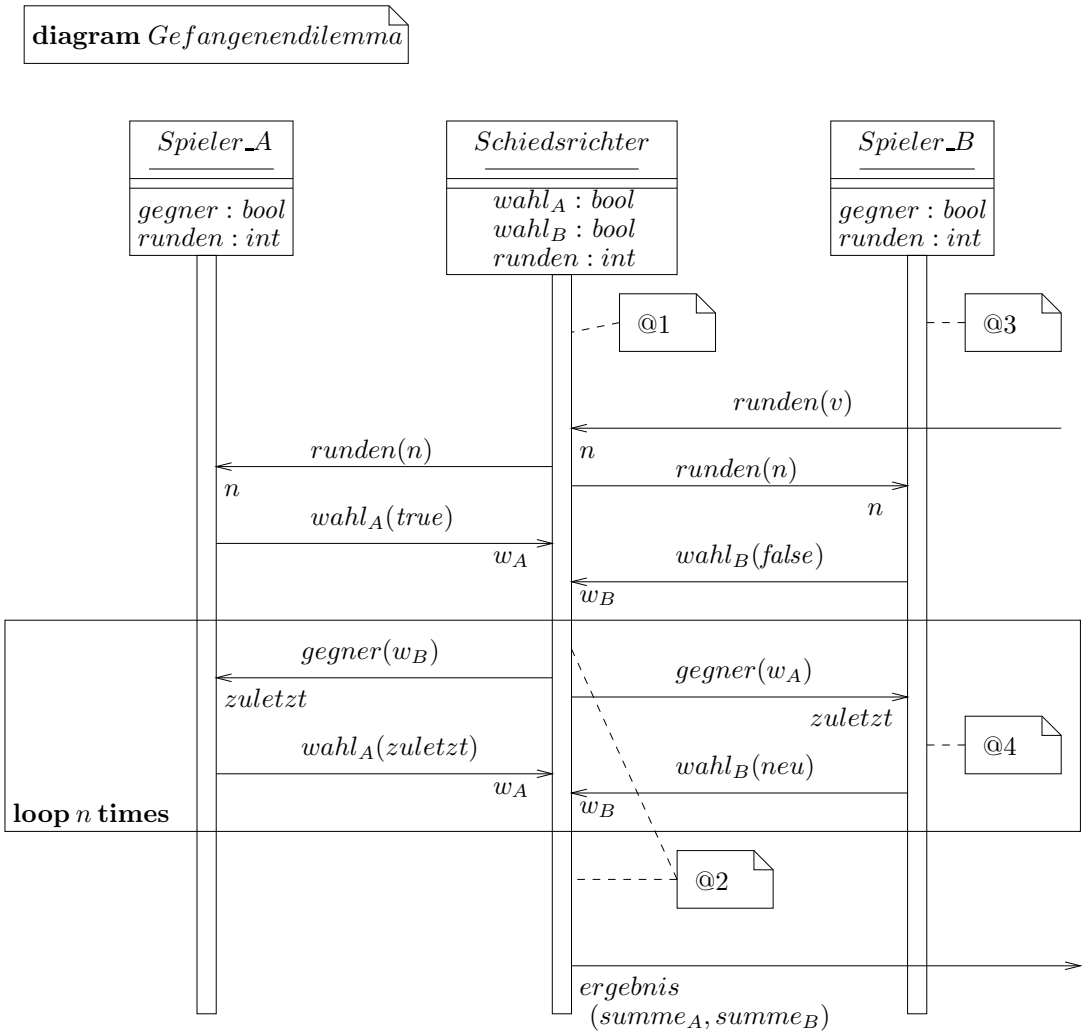


Abbildung 5.18: Spezifikation des Gefangenendilemmas.

Spieler A bzw. B erhält, sowie das Tor *runden*, über das die Umgebung die Anzahl der durchzuführenden Spielzüge mitteilt. Die Tore $wahl_A$ und $wahl_B$ sind vom Typ *bool gate*; der Wert *true* steht für Kooperation und *false* für Betrug. Die beiden Spieler-Instanzen werden in jeder Runde über das Tor *gegner* über den jeweils vorhergehenden Zug ihres Gegners informiert; über das Tor *runden* erhalten sie vom Schiedsrichter die Anzahl der zu spielenden Runden.

Zu Beginn wird in der Instanz *Schiedsrichter* eine Funktion *auswerten* für die Auswertung einer Spielrunde deklariert und die Punktekonten für beide Spieler auf 0 gesetzt (Eintrag @1 im Datenverzeichnis). Danach erhält der Schiedsrichter die Anzahl der Runden aus der Umgebung des Systems. Dieser Wert wird den Spielern mitgeteilt. Anschließend erhält der Schiedsrichter den jeweils ersten Zug beider Spieler. In jedem Schleifendurchlauf werden zuerst die Punktekonten an-

@1	$auswerten =$ $\lambda x : bool.\lambda y : bool.$ $\quad if\ x\ and\ y\ then\ 3$ $\quad else\ if\ x\ and\ not\ y\ then\ 5$ $\quad else\ if\ not\ x\ and\ y\ then\ 0$ $\quad else\ if\ not\ x\ and\ not\ y\ then\ 1$ $summe_A = 0$ $summe_B = 0$	@2	$summe_A =$ $\quad summe_A + auswerten\ w_A\ w_B$ $summe_B =$ $\quad summe_B + auswerten\ w_B\ w_A$
@3	$zuege = []$ $anzahl =$ $\quad let\ anzahl_0 =$ $\quad\quad \lambda f : (bool\ list \to int) \to bool\ list \to int.$ $\quad\quad \lambda l : bool\ list.$ $\quad\quad\quad if\ l = []\ then\ 0\ else\ 1 + f(tail\ l)$ $\quad in\ Y\ anzahl_0$ $betrogen =$ $\quad let\ betrogen_0 =$ $\quad\quad \lambda f : (bool\ list \to int) \to bool\ list \to int.$ $\quad\quad \lambda l : bool\ list.$ $\quad\quad\quad if\ l = []\ then\ 0$ $\quad\quad\quad else\ if\ head\ l\ then\ f(tail\ l)$ $\quad\quad\quad\quad\quad\quad else\ 1 + f(tail\ l)$ $\quad in\ Y\ betrogen_0$	@4	$zuege = cons\ zuletzt\ zuege$ $neu =$ $\quad let\ b = betrogen\ zuege$ $\quad\quad in\ not(anzahl\ zuege - b < b)$

Abbildung 5.19: Datenverzeichnis des Gefangenendilemmas.

hand der jeweils vorherigen Züge mit Hilfe der Funktion *auswerten* aktualisiert. Dann informiert der Schiedsrichter die beiden Spieler über den vorherigen Zug ihres Gegners und empfängt die neuen Spielzüge der Spieler. Nach Beendigung der Schleife werden die beiden Punktekonten noch einmal anhand der letzten Züge aktualisiert und anschließend an die Systemumgebung gesendet.

Die Strategie von Spieler *A* wird *Tit for Tat* (etwa: *Wie du mir, so ich dir*) genannt[6]. In der ersten Runde kooperiert der Spieler. In jeder weiteren Runde wiederholt der Spieler den jeweils vorhergehenden Zug des Gegners. Der zweite Spieler merkt sich in einer Liste die bisherigen Entscheidungen des Gegners. In der ersten Runde betrügt er; in jeder weiteren Runde überprüft er, ob sein Gegner bisher öfter kooperiert oder betrogen hat. Wurde öfter betrogen, betrügt auch der Spieler, wurde öfter kooperiert, kooperiert auch der Spieler. Zur Berechnung von der Anzahl der bisherigen Züge und der Anzahl der Entscheidungen für Betrug des Gegners verwendet *Spieler_B* die rekursiven Funktionen *anzahl* bzw. *betrogen* (siehe Eintrag @3 im Datenverzeichnis). Diese Funktionen werden dann im Eintrag @4 des Datenverzeichnisses zur Berechnung des aktuellen Spielzugs

[6]Es hat sich gezeigt, daß diese Strategie in Turnieren mit vielen Spielen im Mittel relativ erfolgreich ist [Hof88].

verwendet.

5.3.6 Konkatenation von Diagrammen.

Wie zuvor erwähnt, verwenden wir die n-Agenten-Diagramme zur *generativen* Spezifikation von Systemabläufen: Die Modelle beschreiben nicht nur einen, sondern eine Menge möglicher Ausführungen. Bei komplexen Systemen kann die Darstellung des Systemablaufs innerhalb eines einzelnen Diagramms zu einer unübersichtlichen Spezifikation führen, da das Diagramm zu groß wird und/oder zu viele graphische Elemente enthält. Weiterhin ist es wünschenswert, die Beschreibung des Systemverhaltens aus einzelnen Bauteilen zusammenzusetzen. Zum einen erlaubt dies die Wiederverwendbarkeit von Spezifikationsteilen, zum anderen kann so komplexes Systemverhalten durch eine Kombination einfacher Diagramme dargestellt werden.

Wir führen in diesem Abschnitt die Konkatenation von Diagrammen ein. Auf diese Weise erlauben wir die Zerlegung der Beschreibung des Systemverhaltens in mehrere Teildiagramme, die dann über ein spezielles Konstrukt zur sequentiellen Komposition zu einer vollständigen Beschreibung des Verhaltens verbunden werden. Hierbei erlauben wir auch die Angabe von Alternativen, so daß eine Menge von möglichen Systemabläufen spezifiziert werden kann. Eine Menge von Diagrammen, die gemeinsam das Verhalten eines Systems beschreiben, nennen wir *Dokument*.

Durch die Möglichkeit zur Komposition wird zum einen die Übersichtlichkeit von Systembeschreibungen erhöht, da die Unterteilung des Systemverhaltens in inhaltlich zusammengehörige Abschnitte als zusätzliche optische Strukturierung einsetzbar ist. Zum anderen erlaubt die Auswahl zwischen mehreren Fortsetzungsmöglichkeiten die Beschreibung von komplexem Systemverhalten (z.B. der Auswahl zwischen Alternativen, die Schleifen beinhalten), das ohne die Möglichkeit zur Konkatenation nicht darstellbar wäre.

Annotationen. Wir kennzeichnen das Diagramm, mit dem innerhalb eines Dokuments die Beschreibung des Systemverhaltens beginnen soll, durch die Angabe des zusätzlichen Schlüsselworts **initial** in der Annotation, die den Namen des Diagramms beinhaltet. Die Angabe des Startdiagramms muß eindeutig sein.

Am Ende eines Diagramms kann in einer Annotation angegeben werden, mit welchem Diagramm die Ausführung des Systems fortgesetzt werden soll. Wird nur ein möglicher Nachfolger D angegeben, enthält die Annotation den Text **continue with** D. Die Angabe mehrere Alternativen geschieht mit

$$\textbf{continue depending on } I \textbf{ if } E_1 \textbf{ then } D_1 \ldots \textbf{if } E_l \textbf{ then } D_l$$

Wie bei der *while*-Schleife wählen wir eine Instanz I aus, die die Entscheidung, mit welchem Diagramm fortgefahren werden soll, in Abhängigkeit seiner

lokalen Daten trifft. Hierzu werden für die l Alternativen entsprechend l Ausdrücke E_1 bis E_l angegeben. Wird der i-te Ausdruck zu dem Wert *true* reduziert, wird die Ausführung mit dem Diagramm D_i fortgesetzt. Ergeben mehrere Ausdrücke den Wert *true*, erfolgt die Auswahl nichtdeterministisch. Die Instanz, die für die Entscheidung verantwortlich ist, informiert die anderen Instanzen durch synchrone Nachrichten über implizite Tore über die Entscheidung. Wie bei der *while*-Schleife nehmen wir weder die Deklaration der impliziten Tore noch die zugehörigen Nachrichten explizit in die Diagramme auf.

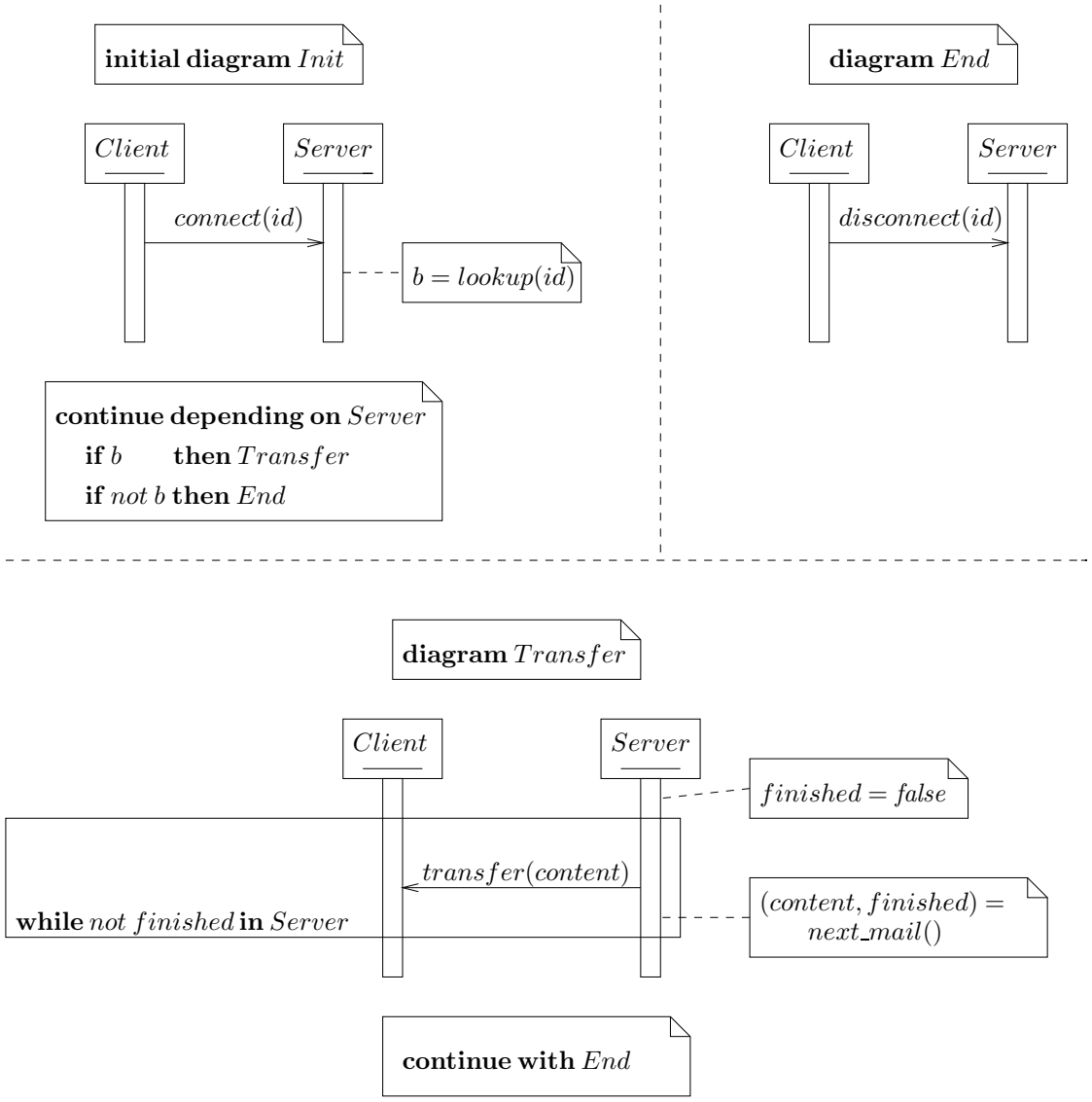


Abbildung 5.20: Diagramme mit Konkatenation.

Beispiel 5.3 *In Abbildung 5.20 ist das Verhalten eines Systems beschrieben, in dem ein* Client *seine elektronische Post von einem* Server *herunterladen will. Die Beschreibung des Verhaltens ist dabei in drei Diagramme zerlegt worden, die durch Konkatenation verbunden sind. Die Ausführung des Systems beginnt mit dem Diagramm* Init*. Nachdem der* Client *sich durch Senden seiner id beim* Server *angemeldet hat, schaut der* Server *nach, ob Post für den* Client *eingegangen ist. Das boolesche Ergebnis dieser Datenoperation wird an den Bezeichner b gebunden. In Abhängigkeit des Wertes von b in* Server *wird dann entschieden, mit welchem Verhalten der Systemablauf fortgesetzt werden soll. Gilt b, ist Post für den* Client *vorhanden und das System fährt mit dem Diagramm* Transfer *fort. Andernfalls wird das Verhalten mit dem Diagramm* End *fortgesetzt. Die Entscheidung wird dem* Client *über implizite Nachrichten mitgeteilt.*

Im Diagramm Transfer *werden die elektronischen Briefe in einer Schleife vom* Server *an den* Client *übertragen. Wir nehmen an, daß die Funktion* next_mail *den nächsten elektronischen Brief sowie die Information, ob der aktuelle Brief der letzte ist, liefert. Anschließend wird die Ausführung des Systems mit dem Diagramm* End *fortgesetzt. Im Diagramm* End *meldet sich der* Client *beim* Server *ab und der Systemablauf endet.*

Parameter. In diesem Beispiel werden bei der Konkatenation von Diagrammen keine Daten über Diagrammgrenzen hinweg verwendet. Für die Verwendung von von Daten über Diagramme hinweg führen wir die Definition von Parametern ein und legen fest, daß nur explizit angegebene Parameter im folgenden Diagramm Gültigkeit besitzen. Würden wir annehmen, daß alle Bezeichner eines Diagramms bei einer Konkatenation im folgenden Diagramm weiterhin gelten, könnte es zu Konflikten bzgl. des Gültigkeitsbereichs von Bezeichnern kommen. Der Grund hierfür ist, daß ein Diagramm mehrere Vorgänger haben kann. Ein Beispiel für einen solchen Konflikt ist in Abbildung 5.21 angegeben. Sowohl das Diagramm D_1 als auch D_2 sind mögliche Vorgänger von D_3. Während in D_1 in der Instanz I_1 ein Bezeichner x deklariert wird, fehlt diese Deklaration in D_2. Daher entsteht in D_3 als Parameter von m ein offener Term, wenn D_2 vor D_3 ausgeführt wird. Aus diesem Grund führen wir die Übergabe von Werten über Parameter ein, da dann gewährleistet ist, daß alle offenen Bezeichner von Diagrammen mit Werten belegt werden. Außerdem unterstützt die Angabe von Parametern die Wiederverwendbarkeit von Diagrammen in anderen Spezifikationen, da auf diese Weise eine Schnittstelle für die Daten zwischen den einzelnen Diagrammen geschaffen wird.

Da Bezeichner nur lokal in den Instanzen auftreten, ist auch die Parameterübergabe zwischen Diagrammen instanzbezogen. Wir deklarieren die Parameter einer Instanz im mittleren Teil der Instanzdeklarationen, in denen sich bei UML die Liste der Attribute befindet (siehe die Instanzen in den Diagrammen *Transfer* und *End* in Abbildung 5.22). Die Instanzen im initialen Diagramm eines

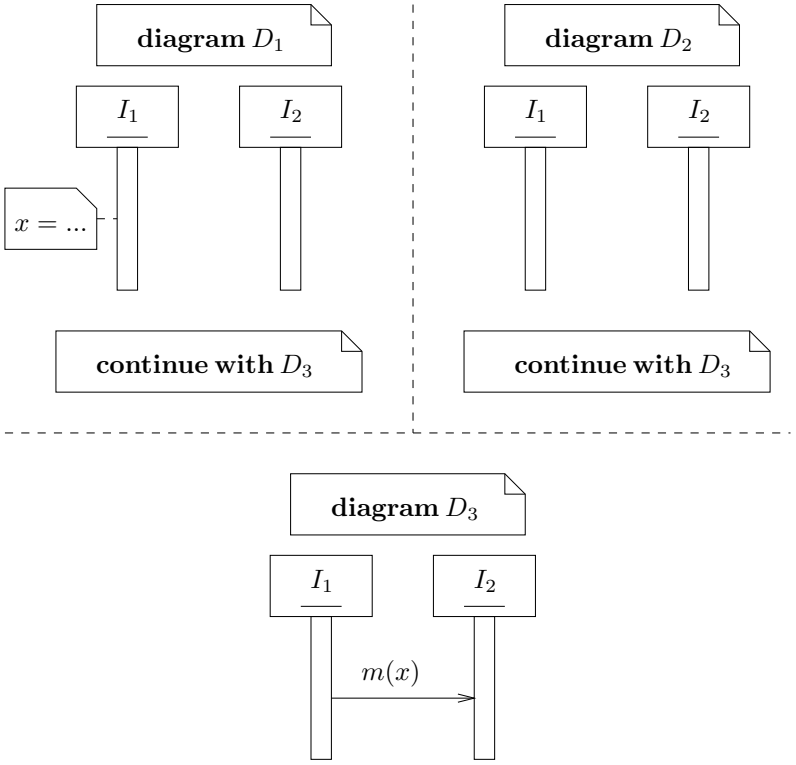


Abbildung 5.21: Konkatenation mit Gültigkeitsbereich-Konflikt.

Dokuments dürfen keine Parameterdeklarationen enthalten.

Wir erweitern die Syntax der Annotation für die Fortsetzung der Systemausführung um Parameterübergabe. Nach der Angabe einer Instanz erlauben wir nach Nennung eines Diagrammnamens die Auflistung der Parameter der einzelnen Instanzen. Diese Liste wird durch das Schlüsselwort **parameters** eingeleitet. Die Syntax für eine unbedingte Fortsetzung mit Parameterübergabe lautet

$$\textbf{continue with } D \textbf{ parameters } I_1(E_{11}, ..., E_{1n_1}), ..., I_k(E_{k1}, ..., E_{kn_k}),$$

die Syntax für die bedingte Fortsetzung entsprechend

continue depending on I
if E_1 **then** D_1 **parameters** $I_1(E_{111}, ..., E_{11n_{11}}), ..., I_k(E_{1k1}, ..., E_{1kn_{k1}})$
...
if E_l **then** D_l **parameters** $I_1(E_{l11}, ..., E_{l1n_{l1}})..., I_k(E_{lk1}, ..., E_{lkn_{kl}})$.

Wir erlauben bei der bedingten Fortsetzung die Angabe individueller Parameter für jede Alternative. Weiterhin muß für jede Instanz die Anzahl der Parameter

beim Aufruf mit der Anzahl der deklarierten Parameter der Instanz im aufgerufenen Diagramm übereinstimmen.

Beispiel 5.4 (Fortsetzung von Beispiel 5.3) *In Abbildung 5.22 ist eine Variante des in Abbildung 5.20 angegeben Mail-Server-Beispiels dargestellt, die um die Übergabe von Daten in Konkatenationsoperatoren erweitert wurde. Zu Beginn der Ausführung im Diagramm* Init *definiert der* Client *einen Bezeichner text, an den die Liste der einzelnen Mails gebunden werden sollen. Mit der Nachricht ask_mail meldet sich der* Client *beim* Server *an und überträgt dabei seine id. Der Server überprüft mit der Funktion look_up, ob Post für den* Client *eingegangen ist. Wir nehmen an, daß look_up einen Tupel aus zwei Werten zurückliefert. Der erste Wert ist ein boolescher Wert, der angibt, ob noch Mail für den* Client *vorhanden ist. Ist dies der Fall, enthält die zweite Komponente die aktuelle Mail in Form einer Zeichenkette; andernfalls ist die Zeichenkette leer. Weiterhin nehmen wir an, daß look_up die jeweils gelesene Mail löscht, so daß bei jedem Aufruf der Funktion eine neue Mail geliefert wird. Ist Post für den* Client *vorhanden, wird die Systemausführung mit dem im Diagramm* Transfer *beschriebenen Verhalten fortgesetzt. Dabei werden im* Client *die beiden Parameter text und id übergeben sowie im* Server *die Parameter id und mail. Ist keine Post vorhanden, wird die Ausführung mit dem Diagramm* End *fortgesetzt. Dabei werden in* Client *ebenfalls die Parameter text und id übergeben, während in* Server *nur id übergeben wird.*

Im Diagramm Transfer *sendet der* Server *die aktuelle Mail mittels* transfer *an den* Client*. Dieser fügt die Zeichenkette mit der Funktion* append *an die Liste der Mails an. Der* Server *überprüft erneut, ob weitere Post für den* Client *vorhanden ist. Ist dies der Fall, wird das Diagramm* Transfer *erneut ausgeführt; andernfalls wird die Ausführung des Systems mit* End *fortgesetzt. In* End *signalisiert der* Client *mit der Nachricht* disconnect *das Ende der Interaktion mit dem* Server*.*

Neben den Datenparametern müssen auch dynamisch erzeugte Tornamen als Parameter an nachfolgende Diagramme übergeben werden, da das neue Tor sonst im nachfolgenden Diagramm nicht sichtbar ist. Dabei muß der entsprechende Bezeichner in der Parameterliste des nachfolgenden Diagramms dem Tornamen im vorangehenden Diagramm entsprechen.

Übersichtsgraph. Der Konkatenationsoperator wird jeweils in den einzelnen Diagrammen angegeben. Hierdurch wird der Kontrollfluß des gesamten Systems nicht explizit graphisch dargestellt, sondern muß implizit aus den Annotationen der einzelnen Diagramme zusammengesetzt werden. Aus diesem Grund führen wir *Übersichtsgraphen* ein, mit denen ein Überblick über das gesamte Systemverhalten angegeben werden kann. Ein Übersichtsgraph wird einem Dokument zugeordnet und ist ein Transitionssystem (siehe Definition 2.1), in dem die Knotenmenge der Menge der Diagramme des Dokuments entspricht. Jeder Knoten wird durch ein Rechteck mit abgerundeten Ecken dargestellt und wird mit dem

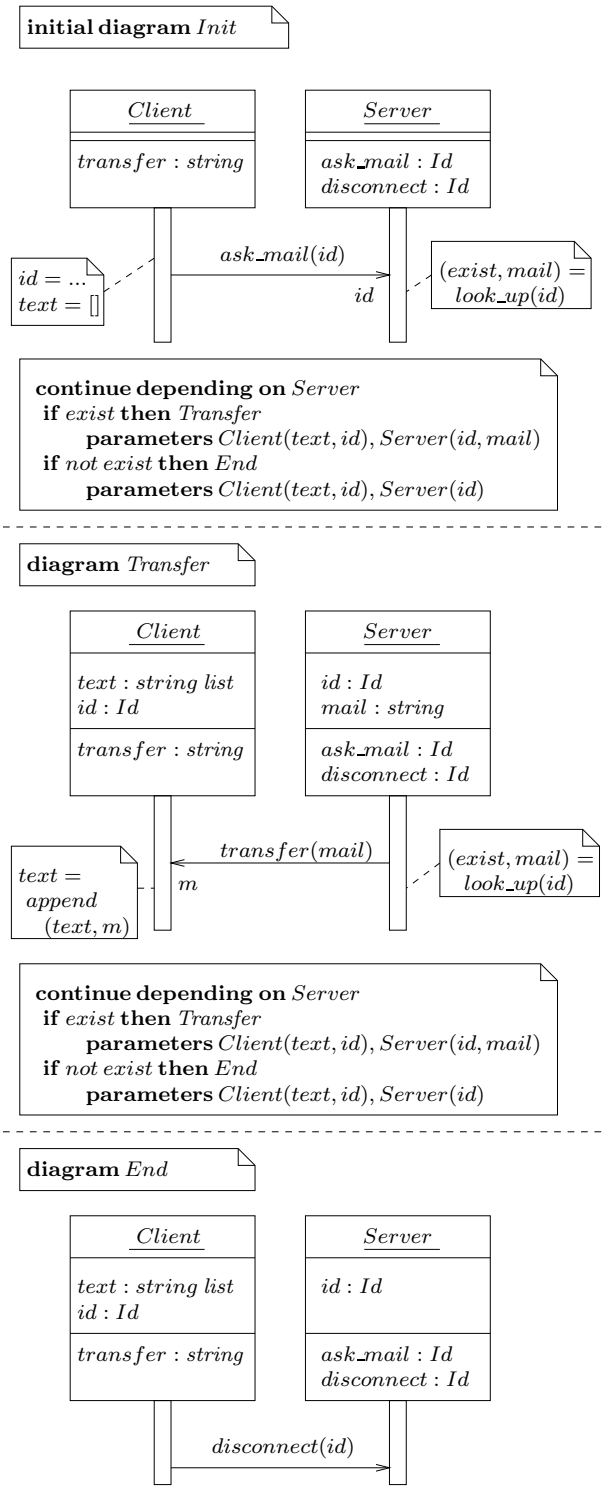


Abbildung 5.22: Konkatenation von Diagrammen mit Daten.

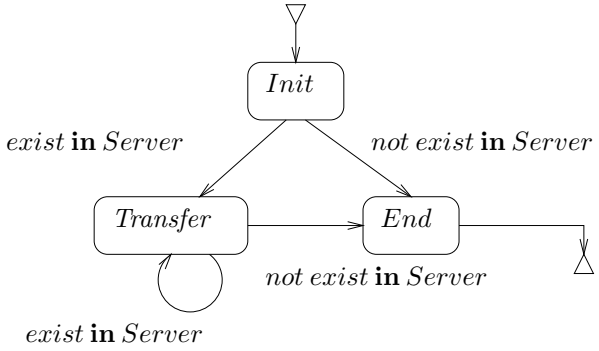


Abbildung 5.23: Übersichtsgraph.

Namen des von ihm repräsentierten Diagramms beschriftet[7]. Zusätzlich verwenden wir die Symbole $\bigtriangledown$ and $\bigtriangleup$ zur Kennzeichnung des Start bzw. Endezustands: Der Knoten, der mit $\bigtriangledown$ durch einen Eingangspfeil verbunden ist, enthält das initiale Diagramm; Knoten, die Diagramme ohne Konkatenationsoperator repräsentieren, werden durch einen Ausgangspfeil mit $\bigtriangleup$ verbunden. Die Transitionsrelation ergibt sich aus den Konkatenationsoperatoren in den Diagrammen. Sind zwei Diagramme durch einen solchen Operator verbunden, werden sie durch einen Transitionspfeil im Übersichtsgraph verbunden. Besitzt ein Knoten mehrere Ausgangspfeile, werden die entsprechenden Bedingungen in der Syntax E **in** I als Beschriftung an die Pfeile geschrieben. Unbedingte Übergänge werden nicht beschriftet. Hinzu kommen die Pfeile von $\bigtriangledown$ zum initialen Zustand und von allen finalen Zuständen zu $\bigtriangleup$.

Beispiel 5.5 (Fortsetzung von Beispiel 5.4) *In Abbildung 5.23 ist der Übersichtsgraph für das Dokument aus Abbildung 5.22 angegeben. Zuerst wird das im Diagramm* Init *beschriebene Verhalten durchgeführt. Anschließend wird in Abhängigkeit des Wertes von exist in der Instanz* Server *die Ausführung mit* Transfer *bzw.* End *durchgeführt. Am Ende von* Transfer *wird entschieden, ob das von* Transfer *beschriebene Verhalten erneut ausgeführt werden soll.* End *ist ein finales Diagramm, so daß der zugehörige Knoten mit dem Endzustand $\bigtriangleup$ verbunden wird.*

Für Übersichtsgraphen legen wir folgende semantische Bedingungen fest:

- $\bigtriangledown$ darf nur mit dem Knoten, der das initiale Diagramm repräsentiert, in Relation stehen.
- Von jedem Knoten darf entweder nur genau ein unbeschrifteter Pfeil oder mehrere beschriftete Pfeile ausgehen.

[7]Bei der Syntax der Diagrammelemente orientieren wir uns an den *High-level Message Sequence Charts* [MR97].

- Gehen von einem Knoten mehrere beschriftete Pfeile aus, müssen alle Beschriftungen denselben Instanznamen beinhalten.

- Knoten für finale Diagramme stehen mit $\triangle$ in Relation.

Bis auf die Angabe der Ausdrücke für die bedingten Transitionen abstrahieren wir in den Übergangsgraphen von den Datentransformationen des modellierten Systems, da Übergangsgraphen ausschließlich zur Darstellung des Kontrollflusses vorgesehen sind.

Die Übersichtsgraphen sind keine Erweiterung der n-Agenten-Diagramme, da sie keine neuen Sprachkonstrukte zur Spezifikation von Kontroll- oder Datenfluß enthalten. Sie dienen nur zur zusätzlichen Visualisierung des Kontrollflusses und können durch eine Analyse der einzelnen Diagramme des entsprechenden Dokuments generiert werden.

Verhältnis von Konkatenation zu Alternativen und Schleifen. In diesem Abschnitt diskutieren wir die Beziehungen zwischen in Abschnitt 5.3.3 und Abschnitt 5.3.4 vorgestellten Alternativen und Schleifen und der Konkatenation von Diagrammen.

Durch die Konkatenation eines Diagramms mit sich selbst ist die wiederholte Ausführung eines Diagramms möglich. Daher lassen sich wiederholte Abläufe auch durch die Konkatenation von Diagrammen realisieren (siehe die Ausführung von *Transfer* in Abbildung 5.22). Bei dieser Darstellung von Schleifen durch Iteration ganzer Diagramme sind stets alle Instanzen an der iterierten Ausführung beteiligt. Hingegen erlauben die Schleifen aus Abschnitt 5.3.4 die Iteration für eine Teilmenge der Instanzen. Weiterhin erlaubt die kompakte Darstellung von Iteration mittels einer *loop*- oder *while*-Schleife eine übersichtliche Form der Darstellung iterierten Verhaltens, bei der auch die Ereignisse im Umfeld der Schleife im Zusammenhang dargestellt werden können.

Ebenso läßt sich die Auswahl zwischen mehreren Alternativen durch die bedingte Konkatenation darstellen. Auch hierbei sind dann alle Instanzen eines Systems an der Durchführung des alternativen Verhaltens beteiligt. Hingegen erlaubt das Konstrukt für alternatives Verhalten aus Abschnitt 5.3.3, besonders in Verbindung mit der alternativen Eingabe von Seite 133, die Beschreibung alternativen Verhaltens innerhalb einer Instanz bzw. in einer Teilmenge der Instanzen. Außerdem läßt sich so die Auswahl zwischen Alternativen, die nur wenige Ereignisse enthalten, übersichtlicher darstellen.

Die Darstellung von komplexem Verhalten mittels einer Schleife bzw. einer Auswahl zwischen Alternativen kann aus Gründen der Übersichtlichkeit insbesondere dann vorteilhaft sein, wenn nur eine Teilmenge der Instanzen an der Durchführung beteiligt ist oder das entsprechende Sprachkonstrukt nur eine kleine Menge von Aktionen beinhaltet. Zudem ist bei dieser Art der Darstellung leichter ersichtlich, welche Instanzen an der Durchführung des Sprachkonstrukts

wirklich beteiligt sind, da bei der Darstellung mittels Konkatenation stets alle Instanzen involviert sind. Enthalten der Schleifenrumpf bzw. die Rümpfe der einzelnen Alternativen viele Aktionen oder sind alle Instanzen des Systems an der Ausführung des komplexen Sprachkonstrukts beteiligt, kann die Verwendung der Konkatenation die übersichtliche Darstellung des Systemverhaltens ermöglichen; insbesondere, wenn zusätzlich zu den einzelnen Diagrammen der Übersichtsgraph des Systemverhaltens angegeben wird.

5.4 Textuelle Notation

Für Message Sequence Charts [ITU96a] wurde neben einer graphischen Darstellung auch einer textuelle Notation für die Systembeschreibung angegeben. Dies hat zum einen den Vorteil, daß Diagramme in Textdateien abgelegt werden können. Zum anderen erlaubt die textuelle Darstellung eine vereinfachte Formulierung der Semantik der Diagramme, da sie die Angabe von Übersetzungsfunktionen unterstützt [MR94a, ITU96b, MR97] (siehe Kapitel 6). In diesem Abschnitt definieren wir eine textuelle Notation zur Darstellung der n-Agenten-Diagramme. Diese Notation ist wie die Notation von MSCs instanzorientiert: Die Lebenslinie einer Instanz wird durch sequentielle Komposition der textuellen Darstellung der Ereignisse angegeben. Die textuelle Darstellung eines Dokuments besteht dann aus einer Menge von textuellen Instanzbeschreibungen. Neben den aus der textuellen Darstellung von MSCs bekannten Konstrukte zur Darstellung des Verhaltens enthält unsere textuelle Notation auch Elemente zur Beschreibung der Datenaspekte, so daß die textuelle Notation alle Elemente der Diagramme abbilden kann.

Zuerst definieren wir die Syntax in Form einer erweiterten kontextfreien Grammatik. Nach der Angabe eines Beispiels für eine textuelle Spezifikation definieren wir semantische Bedingungen für korrekte Spezifikationen.

5.4.1 Syntax

In Abbildung 5.24 wird die textuelle Notation in Form einer erweiterten kontextfreien Grammatik definiert. Hierbei sind in eckige Klammern eingerahmte Teile von Produktionen optional; die geschweiften Klammern entsprechen dem Kleene-Stern: Zeichenfolgen in geschweiften Klammern treten beliebig oft (eventuell auch gar nicht) auf [Sch92]. Wir bezeichnen die von dem Nichtterminalsymbol x erzeugte Sprache mit $L(x)$.

Diagramme und Instanzen. Ein Dokument besteht aus einer nicht-leeren Menge von Diagrammen. Ein Diagramm besteht aus dem Kopf und einer nicht-leeren Menge von Instanzendeklarationen. Der Diagrammkopf besteht aus dem

document	::=	*diagram* { *diagram* }
diagram	::=	[**initial**] **diagram** *d_id instance_list*
instance_list	::=	*instance* { *instance* }
instance	::=	**instance** *i_id* [**uses** *param_list*] [**gates** *gate_list*]
		events *events* [*continue*] **end instance**
param_list	::=	*ident* **:** *type* { **;** *ident* **:** *type* }
gate_list	::=	*m_id* **:** *type* { **;** *m_id* **:** *type* }
events	::=	*event* { **;** *event* }
event	::=	**skip** | *comm_event* | *data_decl* | *chan_decl* | *choice* |
		input_choice | *loops*
input_choice	::=	**choose** *in_event* { **or** *in_event* }
choice	::=	**select** *alternative* { *alternative* } **end select**
alternative	::=	**if** *expr events*
loops	::=	**loop** *ident* **times instances** *id_list* **do** *events* **end loop** |
		while *expr* **in** *i_id* **instances** *id_list* **do** *events* **end while**
id_list	::=	*i_id* { **,** *i_id* }
comm_event	::=	*out_event* | *in_event*
out_event	::=	**out** *m_id* (*expr_list*)
		[**call** | **return**] [**asynch**]
		to *address*
in_event	::=	**in** *m_id* (*expr_list*)
		[**call** | **return**]
		from *address* [**bind** *ident_list*]
address	::=	*i_id* | **env**
ident_list	::=	*ident* { **,** *ident* }
expr_list	::=	[*expr* { **,** *expr* }]
data_decl	::=	**data** *decl* { **,** *decl* }
chan_decl	::=	**gate** *gate_list*
decl	::=	*ident* **=** *expr*
continue	::=	**continue with** *d_id* [*params*] |
		continue depending on *i_id*
		if *expr* **then** *d_id* [*params*]
		{ **if** *expr* **then** *d_id* [*params*] }
params	::=	**parameters** *i_id* (*expr_list*) { **,** *i_id* (*expr_list*) }

Abbildung 5.24: Textuelle Notation für Sequenzdiagramme.

Schlüsselwort **diagram**, gefolgt von einem Diagrammnamen $d \in L(d\_id)$. Optional kann das Diagramm durch ein vorangehendes **initial** als Beginn eines Systemverhaltens bestehend aus mehreren Diagrammen gekennzeichnet werden. Die Deklaration einer Instanz besteht aus drei Teilen: Nach dem Bezeicher $i \in L(i\_id)$ wird optional nach dem Schlüsselwort **uses** die Liste der Datenparameter für die Konkatenation von Diagrammen angegeben. Hierauf folgen nach dem Schlüsselwort **gates** die Tore der Instanz. Nach dem Schlüsselwort **events** folgen die einzelnen Ereignisse $e \in L(event)$. Mögliche Ereignisse sind das leere Ereignis **skip**, Kommunikationsereignisse, Datentransformationen, die Auswahl zwischen mehreren Alternativen sowie Schleifen. Die Folge von Ereignissen kann optional durch einen Konkatenationsoperator aus $L(continue)$ abgeschlossen werden.

Ereignisse. Zu den Kommunikationsereignissen gehören das Senden einer Nachricht $m \in L(m\_id)$ sowie das Empfangen einer Nachricht. Außer an andere Instanzen kann eine Nachricht auch an die Umgebung des Sequenzdiagramms gesendet bzw. von dort empfangen werden. Die Umgebung wird mit **env** adressiert. Die Parameter einer Nachricht müssen sowohl bei der Sendeaktion als auch bei der Empfangsaktion angegeben werden. Asynchrone Nachrichten werden mit dem optionalen Schlüsselwort **asynch** gekennzeichnet. Pfeile mit ausgefüllten Spitzen, die Anfragen symbolisieren, werden mit dem optionalen Schlüsselwort **call** gekennzeichnet, gestrichelte Antwortnachrichten entsprechend mit **return**. Für die Kurzform von Anfragen (siehe Abbildung 5.9) geben wir keine spezielle textuelle Form an; wir nehmen an, daß die Kurzform vor der Umsetzung in die textuelle Darstellung expandiert wird. Die Auswahl zwischen dem Empfang mehrerer Instanzen einer Nachricht wird durch das Schlüsselwort **choose** eingeleitet, auf das die verschiedenen Eingabeaktionen, getrennt durch das Schlüsselwort **or**, folgen. Hierbei muß mindestens ein Eingabeereignis angegeben werden. Die Angabe von Bezeichnern zur Bindung der empfangenen Werte muß dabei in jedem der Eingabeereignisse angegeben werden.

Werden bei der Empfangsaktion Bezeichner angegeben, an die die empfangenen Werte gebunden werden sollen, wird die Liste der Bezeichner nach dem Schlüsselwort **bind** angegeben.

Die Auswahl zwischen verschiedenen Alternativen wird mit dem Schlüsselwort **select** eingeleitet. Hierauf folgen, jeweils nach dem Schlüsselwort **if**, die Eingangsbedingung und die Ereignisse der entsprechenden Alternative. Treten in der Lebenslinie einer Alternative keine Ereignisse auf, wird als einziges Ereignis das leere Ereignis **skip** angegeben. Abgeschlossen wird die Auswahl mit **end select**.

loop–Schleifen werden innerhalb einer Instanz mit **loop** *id* **times instances** *id_list* **do** eingeleitet, wobei der Wert des Bezeichners *id* die Anzahl der Schleifendurchläufe angibt und *id_list* eine Liste der an der Schleife beteiligten Instanzen definiert. Abgeschlossen werden Schleifen mit **end loop**. Analog werden *while*–

Schleifen mit **while** E **in** I **instances** *id_list* **do** begonnen und mit **end while** beendet.

Am Ende der Instanzen kann angegeben werden, mit welchem Diagramm die Ausführung des Systems fortgesetzt werden soll. Die Syntax $L(continue)$ entspricht dem Inhalt der Annotation in Abschnitt 5.3.6. Wir geben die Konkatenation mit anderen Diagrammen am Ende jeder Instanz an, da die Konkatenation eine Aktion ist, die von jeder Instanz des Systems aktiv durchgeführt wird (im Gegensatz zur **diagram**-Deklaration, die keine Aktionen der Instanzen beinhaltet und daher einmal zu Beginn global angegeben wird).

Daten. Die Inhalte der Datenannotationen werden an der Stelle, an der sie durch die gestrichelte Linie mit der Lebenslinie der jeweiligen Instanz verbunden sind, nach dem Schlüsselwort **data** eingefügt. Analog wird die dynamische Erzeugung von Kanälen mit dem Schlüsselwort **gate** eingeleitet. Sind Daten in einem Datenverzeichnis angegeben, wird der entsprechende Eintrag des Verzeichnisses nach **data** angegeben, da eine spezielle textuelle Darstellung des Datenverzeichnisses nicht vorgesehen ist. Da die Datensprache nicht festgelegt ist, repräsentiert $L(expr)$ die Ausdrücke der jeweils gewählten Datensprache.

Beispiel 5.6 (Fortsetzung des Gefangenendilemmas) *Als Beispiel für eine textuelle Darstellung eines Sequenzdiagramms mit Daten übertragen wir das in Abschnitt 5.3.5 beschriebene Spiel des Gefangenendilemmas.*

diagram *Gefangenendilemma*

instance *Spieler_A*
gates
gegner : *bool*;
runden : *int*
events
in *runden*(n) **from** *Schiedsrichter* **bind** n;
out $wahl_A$(*true*) **to** *Schiedsrichter*;
loop n **times instances** *Spieler_A*, *Spieler_B*, *Schiedsrichter* **do**
in *gegner*(w_B) **from** *Schiedsrichter* **bind** *zuletzt*;
out $wahl_A$(*zuletzt*) **to** *Schiedsrichter*
end loop
end instance

instance *Schiedsrichter*
gates
$wahl_A$: *bool*;
$wahl_B$: *bool*;
runden : *int*

```
events
    data
        auswerten =
            λx : bool.λy : bool.
                if x and y then 3
                else if x and not y then 5
                else if not x and y then 0
                else if not x and not y then 1,
        summe_A = 0,
        summe_B = 0;
    in runden(v) from env bind n;
    out runden(n) to Spieler_A;
    out runden(n) to Spieler_B;
    in wahl_A(true) from Spieler_A bind w_A;
    in wahl_B(false) from Spieler_B bind w_B;
    loop n times instances Spieler_A, Spieler_B, Schiedsrichter do
        data
            summe_A = summe_A + auswerten w_A w_B,
            summe_B = summe_B + auswerten w_B w_A;
        out gegner(w_B) to Spieler_A;
        out gegner(w_A) to Spieler_B;
        in wahl_A(zuletzt) from Spieler_A bind w_A;
        in wahl_B(⟨neu⟩) from Spieler_B bind w_B
    end loop;
    data
        summe_A = summe_A + auswerten w_A w_B,
        summe_B = summe_B + auswerten w_B w_A;
    out ergebnis(summe_A,summe_B) to env
end instance

instance Spieler_B
gates
    gegner : bool;
    runden : int
events
    data
        zuege = [],
        anzahl =
            let anzahl_0 =
                λf : (bool list → int) → bool list → int.
                λl : bool list.
                    if l = [] then 0 else 1 + f(tail l)
                in Y anzahl_0,
```

```
        betrogen =
            let betrogen_0 =
                λf : (bool list → int) → bool list → int.
                λl : bool list.
                    if l = [] then 0
                    else if head l then f(tail l) else 1 + f(tail l)
                in Y betrogen_0;
    in runden(n) from Schiedsrichter bind n;
    out wahl_B(false) to Schiedsrichter;
    loop n times instances Spieler_A, Spieler_B, Schiedsrichter do
        in gegner(w_A) from Schiedsrichter bind zuletzt;
        data
            zuege = cons zuletzt zuege,
            neu =
                let b = betrogen zuege
                    in not(anzahl zuege − b < b);
        out wahl_B(neu) to Schiedsrichter
    end loop
end instance
```

5.4.2 Statische semantische Anforderungen

Die textuelle Notation der Diagramme ist instanzorientiert: Die einzelnen Ereignisse auf der Lebenslinie einer Instanz werden durch eine sequentielle Komposition der entsprechenden textuellen Repräsentationen dargestellt. Während in der graphischen Darstellung die Beziehungen zwischen den einzelnen Instanzen durch z.B. Nachrichten oder Teilnahme in Schleifen direkt ersichtlich sind, ist dies bei der textuellen Notation nur indirekt durch eine Analyse der Instanzbeschreibungen möglich. Daher stellen wir, neben den den bereits in den vorherigen Abschnitten genannten semantischen Anforderungen an die Diagramme, zusätzliche semantische Forderungen an die textuelle Repräsentation zur Vermeidung von Inkonsistenzen. Diese Anforderungen sind statischer Natur, da sie nicht von einem dynamischen Ablauf des spezifizierten Systems abhängen, sondern für alle beschriebenen Abläufe erfüllt sein müssen.

- Die Eingabeereignisse **in** $m(E_1, ..., E_n)$ **from** ... enthalten die Liste der Nachrichtenparameter $E_1, ..., E_n$. Die in den Parametern enthaltenen Bezeichner sind nicht in der empfangenden Instanz gültig.

- Zu jeder Nachricht muß es ein eindeutiges Sende- und ein eindeutiges Empfangsereignis geben. Hierbei muß die Adressierung der Instanzen korrekt angegeben sein, d.h. daß Sendeereignis und das Empfangsereignis zu einer

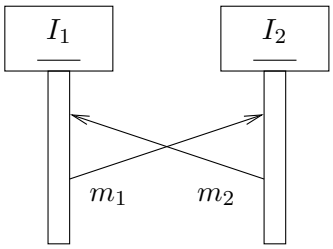


Abbildung 5.25: Semantisch falscher Nachrichtenfluß.

Nachricht müssen korrespondierende Adressen verwenden. Wird eine Nachricht m von einer Instanz I_1 gesendet und von einer Instanz I_2 empfangen, muß das Ereignis **out** $m(...)$ **to** I_2 in I_1 vorkommen und entsprechend **in** $m(...)$ **from** I_1 in I_2.

- Zu jeder Anfrage muß eine zugehörige Antwort existieren, die an den Sender der Anfrage gerichtet ist.
- Es dürfen keine Nachrichtenflüsse angegeben werden, die die Kausalität verletzen. Der folgende Nachrichtenaustausch zwischen zwei Instanzen ist ein Beispiel für eine Kausalitätsverletzung:

I_1:	I_2:
in $m_2()$ **from** I_2**;**	**in** $m_1()$ **from** I_1**;**
out $m_1()$ **to** I_2	**out** $m_2()$ **to** I_1

Der Nachrichtenfluß wird in Abbildung 5.25 als Diagramm dargestellt. Wie im Diagramm leicht zu erkennen ist, ist der dargestellte Nachrichtenaustausch kausal nicht möglich. Die Angabe eines solchen Nachrichtenflusses ist in Diagrammen nicht möglich, da wir festgelegt haben, daß diagonal nach oben gehende Pfeile nicht erlaubt sind (siehe Seite 122). In der textuellen Darstellung werden Sende- und Empfangsereignisse unabhängig voneinander angegeben, so daß der Ausschluß von semantisch falschen Nachrichten wie im Beispiel explizit gefordert werden muß.

- Kontrollstrukturen müssen korrekt verschachtelt sein. Während in den Diagrammen die korrekte Schachtelung durch die Syntax in Form von Rechtecken gewährleistet ist, wird in der instanzbezogenen textuellen Darstellung Schleifen lediglich die lokale Einbindung der jeweiligen Instanz in eine Schleife dargestellt. Daher muß hier, analog zur Gewährleistung eines semantisch korrekten Nachrichtenflusses, die korrekte Schachtelung der Schleifen als zusätzliche semantische Forderung gestellt werden.

- In der Darstellung einer Schleife müssen alle beteiligten Instanzen dieselbe Instanzliste verwenden.

5.5 Diskussion

In diesem Kapitel haben wir das Systemmodell der n-Agenten-Systeme eingeführt, in denen ein System aus parallel ablaufenden Instanzen besteht, die über lokale Datenspeicher verfügen und die über den Austausch von Nachrichten interagieren. Zur graphischen Spezifikation dieser Systeme haben wir die n-Agenten-Diagramme definiert. Diese Diagramme gehören zu den Interaktionsdiagrammen. Ihre Syntax basiert auf den Sequenzdiagrammen von UML [BRJ98] von Booch, Jacobson und Rumbaugh, die Semantik der graphischen Elemente wurde aber an die Verwendung zur Spezifikation verteilter Systeme angepaßt. Zusätzlich wurden die Diagramme um die Beschreibung von Datentransformationen erweitert. Außerdem werden komplexe Sprachelemente wie Schleifen und die Auswahl von Alternativen an das Systemmodell verteilter Instanzen und die Behandlung von Daten angepaßt. Die Auswahloperation trifft ihre Entscheidung nur auf der Basis der lokalen Daten der betreffenden Instanz, so daß die Semantik der Operation eindeutig ist. Bei Schleifen wird durch die Angabe einer Kontrollinstanz oder durch die Bindung von Werten an bestimmte Bezeichner die Eindeutigkeit der Semantik gewährleistet. Weiterhin haben wir den Begriff der Konkatenation von Diagrammen eingeführt, die die Zerlegung der Beschreibung des Systemverhaltens in eine Menge von Diagrammen erlaubt. Die einzelnen Diagramme können dann durch einen speziellen Konkatenationsoperator miteinander verbunden werden, bei dem die Übergabe von Daten möglich ist. Der Operator erlaubt die Angabe mehrerer Alternativen; durch die Festlegung einer Kontrollinstanz ist, wie bei der *while*-Schleife, die Verantwortlichkeit für die Auswahl definiert, so daß die Semantik der bedingten Konkatenation eindeutig ist.

Neben der graphischen Darstellung erlauben wir auch die textuelle Spezifikation des Systemverhaltens durch die Angabe einer textuellen Notation. Diese Darstellung des Systemverhaltens erleichtert die Angabe einer formalen Semantik durch die Definition von Übersetzungsfunktionen (siehe Kapitel 6).

Im folgenden vergleichen wir die n-Agenten-Diagramme mit den verwandten Ansätzen der Message Sequence Charts [ITU96a, ITU99] und der Sequenzdiagramme aus UML [BRJ98, OMG99].

5.5.1 Sequenzdiagramme

UML enthält zur Beschreibung der Interaktion zwischen Instanzen die Sequenzdiagramme und die Kollaborationsdiagramme (siehe Abschnitt 5.1). Während die Kollaborationsdiagramme den Schwerpunkt der Darstellung auf die Beziehungen

zwischen den interagierenden Instanzen setzen, stellen Sequenzdiagramme den zeitlichen Ablauf der Interaktion dar. Die Syntax unserer n-Agenten-Diagramme orientiert sich an der Syntax der Sequenzdiagramme, um die aufgrund der Popularität von UML große Anzahl von Werkzeugen und Zeichenprogrammen nutzen zu können.

Es existiert keine standardisierte textuelle Darstellung von Sequenzdiagrammen. Arbeiten zur Angabe einer Semantik von Sequenzdiagrammen verwenden deshalb oft eigene Notationen, die dann als Grundlage für die Angabe von Übersetzungsfunktionen fungieren (z.B. in [Stö99]). Dies erschwert den Vergleich von Semantiken. MSCs und die n-Agentdiagramme verfügen hingegen über eine standardisierte textuelle Darstellung, so daß hier eine einheitliche textuelle Repräsentation von Diagrammen gegeben ist.

Systemmodell. Während die n-Agenten-Diagramme für die Spezifikation des Verhaltens des speziellen Typs der n-Agenten-Systeme vorgesehen sind, werden bei den Sequenzdiagrammen von UML keine Annahmen über das zugrundeliegende System gemacht. Daher können Sequenzdiagramme sowohl für verteilte Systeme mit parallelen Komponenten als auch für prozedurale, sequentielle Systeme verwendet werden. Es zeigt sich allerdings, daß die Kontrollstrukturen eher für die Modellierung sequentieller, objektorientierter Systeme geeignet sind (s.u.).

Daten. UML erlaubt eine detaillierte Beschreibung der Daten eines Systems durch Verwendung von Klassendiagrammen. Ebenso kann das Verhalten von Systemen durch die dynamischen Modelle dargestellt werden. Allerdings enthält die Beschreibung von UML nur wenig Informationen über die Integration von Daten- und Verhaltensspezifikation.

So ist in Sequenzdiagrammen keine explizite Darstellung von Datentransformationen vorgesehen. Daten treten i. allg. nur in den Parametern von Nachrichten auf. Werden Datentransformationen in Annotationen angegeben, besitzen diese nur den Status von Kommentaren. Da Sequenzdiagramme im Rahmen von UML bevorzugt zur Darstellung des Verhaltens objektorientierter Systeme verwendet werden, entsprechen die Nachrichten meist Methodenaufrufen. Die Typen der Parameter der Methoden werden daher oft in einem zusätzlichen Klassendiagramm festgelegt. Eine explizite Typisierung von Nachrichten, die der Deklaration der Tore aus den n-Agenten-Diagrammen vergleichbar wäre, ist nicht vorgesehen. Im Gegensatz zu den n-Agenten-Diagrammen werden auch keine Informationen bzgl. des Gültigkeitsbereichs von Bezeichnern gegeben. Dies erschwert zum einen die Analyse von Systemen, zum anderen ist so eine automatische Erzeugung von Programmcode aus einer integrierten Daten- und Verhaltensspezifikation nicht ohne weiteres möglich.

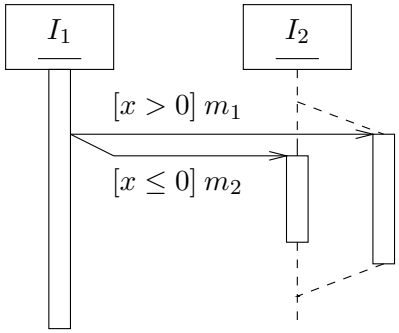


Abbildung 5.26: Alternative Nachrichten im UML.

Alternativen und Schleifen. Zur Darstellung von alternativem Verhalten steht in Sequenzdiagrammen nur die Angabe bedingter Nachrichten wie in Abbildung 5.12 zur Verfügung. Werden bedingte Nachrichten zu derselben Instanz geschickt, kann die alternative Ausführung dieser Instanz durch eine Aufteilung der Lebenslinie modelliert werden (siehe Abbildung 5.26). Wie bereits in Abschnitt 5.3.3 diskutiert, ist dieses Sprachkonstrukt eher für die Spezifikation prozeduraler Systeme geeignet, um bedingte Prozedur- bzw. Methodenaufrufe darzustellen. Bei der Verwendung zur Modellierung nebenläufiger Systeme ist die Semantik, wie in Abschnitt 5.3.3 erläutert, nicht immer eindeutig.

Schleifen werden in [BRJ98, OMG99] nur kurz erwähnt. Es wird nur festgelegt, daß die Schleifenbedingungen am unteren Ende des Rechtecks, das den Schleifenrumpf repräsentiert, angegeben werden sollen. Aussagen über Gültigkeitsbereiche werden, wie bei den Alternativen, nicht gemacht. Weiterhin wird nicht festgelegt, welche Instanz in einem verteilten System für die Steuerung der Schleife verantwortlich ist und wie die anderen Instanzen Informationen von dieser Kontrollinstanz erhalten.

Im Gegensatz hierzu verfügen die n-Agentendiagramme über semantisch eindeutige Konstrukte zur Modellierung von Auswahl und von Schleifen. Ebenso wird der Gültigkeitsbereich von Bezeichnern in Verbindung mit diesen Konstrukten festgelegt. Allerdings sind n-Agenten-Diagramme speziell für verteilte Systeme entworfen und unterstützen nicht explizit die Spezifikation sequentiellen, objektorientierten Verhaltens.

Konkatenation von Diagrammen. Eine Beschreibung eines komplexen Systemverhaltens durch die Zerlegung der Spezifikation in einzelne Teildiagramme, die durch spezielle Operatoren verbunden werden, ist in Sequenzdiagrammen nicht vorgesehen. Durch die Angabe einer Menge von Sequenzdiagrammen ist die Aufteilung der Beschreibung des Systemverhaltens auf verschiedene Diagramme zwar möglich; es gibt aber kein syntaktisches Konstrukt zur Konkatenation von Diagrammen. Eine Möglichkeit besteht darin, die Konkatenation von Diagram-

men informell in Form von Kommentaren anzugeben [GF99]. Bei dieser Methode ist aber, im Gegensatz zu den n-Agenten-Diagrammen, keine formale Semantik für die Konkatenation gegeben, so daß z.B. Fragen nach der Gültigkeit von Bezeichnern über Diagrammgrenzen hinweg offen bleiben.

5.5.2 Message Sequence Charts

Message Sequence Charts (MSC) ist eine weit verbreitete Form von Interaktionsdiagrammen. MSC wurde von der International Telecommunication Union (ITU) standardisiert. Der bisherige Standard MSC '96 [ITU96a] wird durch eine neue Version MSC 2000 [ITU99] abgelöst. Wir diskutieren das Verhältnis von n-Agenten-Diagrammen zu beiden Versionen des MSC-Standards, indem wir einzelne Aspekte der n-Agentendiagramme mit den entsprechenden Konzepten der MSC vergleichen.

Systemmodell. In MSCs ist Kommunikation grundsätzlich asynchron. Jede Instanz besitzt ihre eigene lokale Zeit, so daß ein Vergleich von Zeitpunkten zwischen verschiedenen Instanzen nicht möglich ist. Synchrone Übertragung von Nachrichten muß durch die Verwendung eines Kommunikationsprotokolls durch den Austausch asynchroner Nachrichten simuliert werden. Wie bei den Sequenzdiagrammen werden keine Annahmen über das zugrundeliegende Systemmodell getroffen.

Durch die Unterscheidung von synchroner und asynchroner Kommunikation erlauben die n-Agentendiagramme eine detailliertere Beschreibung des Nachrichtenaustauschs. Zudem beinhalten viele Programmiersprachen zur Implementierung nebenläufiger Systeme wie z.B. *Ada* [TDT95] und *Occam* [Sch88, Ste88] synchrone Kommunikation, so daß Spezifikationen in Form von n-Agenten-Diagrammen direkt in Programmcode dieser Sprachen überführt werden können.

Daten. Im Standard MSC '96 ist eine explizite Behandlung von Datentransformationen nicht vorgesehen. Es besteht lediglich die Möglichkeit, Nachrichten mit Parametern zu versehen und Daten informell in die Beschreibung lokaler Aktionen zu integrieren. Regeln für die Definition und Gültigkeit von Bezeichnern sind nicht festgelegt.

In MSC 2000 [ITU99] ist eine Beschreibung von Daten enthalten. Der Standard unterscheidet zwischen *statischen* und *dynamischen* Daten. Statische Daten sind innerhalb eines gesamten Diagramms in allen Instanzen bekannt und können während der Ausführung des Diagramms nicht verändert werden. Dynamische Daten werden während der Ausführung eines Diagramms innerhalb einer Instanz erzeugt und sind auch nur innerhalb dieser Instanz gültig, sofern sie nicht durch Kommunikation anderen Instanzen bekanntgemacht werden. Wie bei den n-Agenten-Diagrammen ist die Datensprache nicht festgelegt. Es werden keine

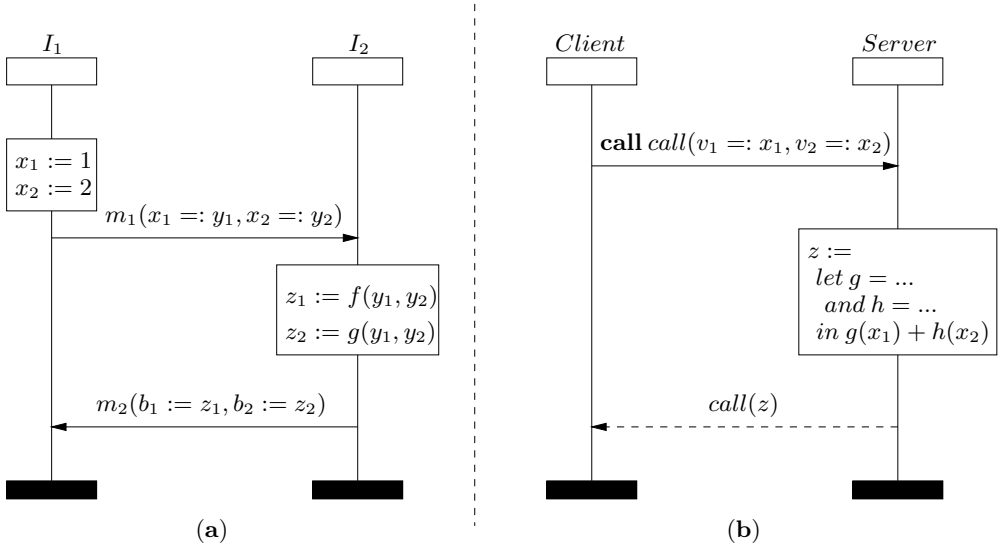


Abbildung 5.27: Message Sequence Chart mit Daten.

Anforderungen an die Datensprache gestellt; im Gegensatz zu den n-Agenten-Diagrammen können neben funktionalen Sprachen auch imperative Sprachen verwendet werden. Allerdings existiert noch keine der bisherigen Semantik [ITU96b] oder der Semantik in Kapitel 6 vergleichbare Semantik für MSCs mit Datenbehandlung.

In Abbildung 5.27(a) ist die MSC 2000-Version des Beispiels aus Abbildung 5.8 angegeben. Die Definition von Bindungen geschieht in den mit Rahmen gekennzeichneten internen Aktionen der Instanzen. Die Bindung der in Nachrichten übertragenen Parameter im Empfänger geschieht durch Angabe der entsprechenden Bezeichner in der Parameterliste: Geht der Pfeil von links nach rechts, wird die Bindung als $v =: x$ geschrieben, geht der Pfeil von rechts nach links, entsprechend $x := v$. In Abbildung 5.27(a) ist zu beachten, daß die Nachrichten im Gegensatz zu Abbildung 5.8 asynchron übertragen werden.

In Abbildung 5.27(b) ist ein MSC mit einem Methodenaufruf angegeben, was einer Anfrage in n-Agenten-Diagrammen entspricht (das Diagramm entspricht der Anfrage aus Abbildung 5.5). Der Aufruf der Methode wird mit dem Schlüsselwort **call** gekennzeichnet, dem der Methodenname und die Parameterliste folgen. Die Rückantwort erhält als Beschriftung ebenfalls den Methodennamen sowie die Liste der Rückgabeparameter.

Eine Deklaration der verwendeten Nachrichten und ihrer Parametertypen ist vorgesehen. Allerdings werden die Nachrichten nicht bestimmten Instanzen zugeordnet, sondern eine Nachricht kann von jeder Instanz gesendet oder empfangen werden, die sich in dem Diagramm befindet, in dem die Deklaration erfolgt. Daher ist die Spezifikation von Zugriffsrechten nicht explizit möglich wie bei der Verwendung der dynamischen Tore der n-Agenten-Diagramme. Zusätzlich können in

MSCs Tore definiert werden, die aber nicht die Nachrichtentypen festlegen, sondern eine Schnittstelle zur Interaktion der Instanzen mit der Systemumgebung darstellen. Sendet in einem Diagramm eine Instanz I eine Nachricht an ein Tor g und empfängt in einem anderen Diagramm eine Instanz J einen Wert über g, können die beiden Diagramme als parallele Ausführung von Systemteilen interpretiert werden, bei der die Nachricht direkt von I an J gesendet wird. Auf diese Weise können Diagramme durch horizontale Komposition "parallelgeschaltet" werden.

Wie bereits in Abschnitt 5.3.3 und Abschnitt 5.3.4 erläutert, wird in MSC 2000 [ITU99] nur rudimentär auf den Zusammenhang zwischen den Gültigkeitsbereichen von Bezeichnern und den Konstrukten zur Modellierung von Alternativen und Schleifen eingegangen. Insbesondere fehlt hierfür eine formale Semantik. In den n-Agentendiagrammen werden Gültigkeitsbereiche in Verbindung mit solchen Sprachkonstrukten erläutert und durch eine formale Semantik definiert (siehe Kapitel 6).

MSC-Dokumente. Der bisherige Standard MSC '96 erlaubt die Zerlegung einer Systembeschreibung in eine Menge von einzelnen MSCs. Hierzu stehen zwei Verfahren zur Verfügung: die *vertikale Komposition* bzw. *alternative Komposition* von MSCs [Ren99] und die Verwendung von High-level MSCs [MR97, Ren99].

Bei der vertikalen Komposition von MSCs wird ein Diagramm an der Unterseite eines anderen Diagramms plaziert. Dann werden die gemeinsam in beiden Diagrammen enthaltenen Instanzen verbunden, so daß die Lebenslinie einer solchen Instanz die sequentielle Komposition des in den einzelnen Diagrammen spezifizierten Verhaltens dieser Instanz darstellt [Ren99]. Die alternative Komposition ist eine Erweiterung der vertikalen Komposition, bei der mehrere mögliche Fortsetzungen des Verhaltens möglich sind. Die Angabe, mit welchem Diagramm unter welchen Bedingungen die Ausführung fortgesetzt werden soll, muß informell zusätzlich zu den Diagrammen erfolgen, da der Standard MSC '96 [ITU96a, Ren99] hierfür keine syntaktischen Elemente bereitstellt.

Eine Möglichkeit zur syntaktischen Darstellung der vertikalen bzw. alternativen Komposition ist die Verwendung von *Bedingungen* (*conditions*), die als "Klebemarken" zwischen einzelnen Diagrammen fungieren [GHRW98]. Bedingungen haben in MSC '96 keine semantische Bedeutung, sondern dienen nur zur informellen Erläuterung des in den MSCs angegebenen Verhaltens. Daher lassen sie sich für die Angabe von Diagrammkomposition verwenden. In Abbildung 5.28 ist ein MSC-Dokument angegeben, daß aus drei Diagrammen besteht. Am Ende des Diagramms D_1 befindet sich eine Bedingung C. Wurde D_1 ausgeführt, kann der weitere Systemablauf mit jedem Diagramm fortfahren, das mit der gleichen Bedingung C beginnt. In Abbildung 5.28 können also D_2 oder D_3 nach D_1 ausgeführt werden. Es ist allerdings nicht klar, wie die Entscheidung über die Auswahl des nachfolgenden Diagramms getroffen wird. In D_2 sendet I_1 ei-

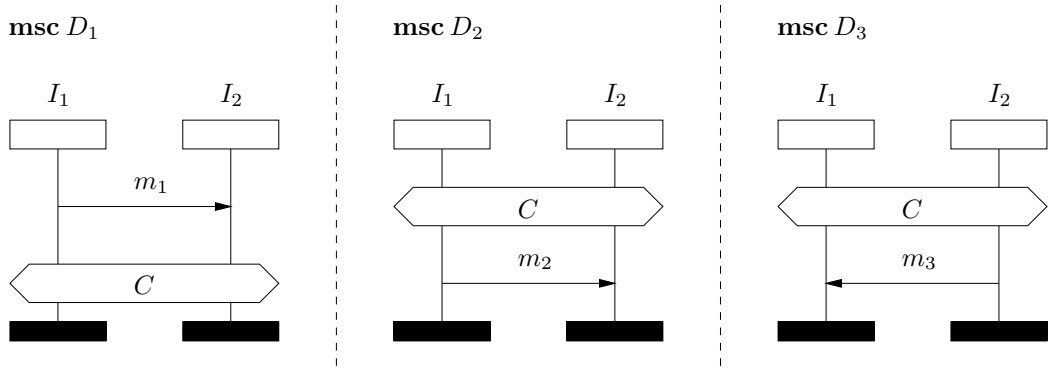


Abbildung 5.28: MSC mit Konkatenation.

ne Nachricht m_1, in D_3 sendet I_2 die Nachricht m_2. Sendet also I_1 zuerst, muß die Ausführung mit dem Diagramm D_2 fortgesetzt werden; sendet hingegen I_2 zuerst, wird entsprechend D_3 ausgeführt. Dies führt dazu, daß, sobald eine Instanz eine Nachricht sendet, die andere Instanz keine eigene Nachricht senden darf; da die Nachrichten asynchron sind, muß die andere Instanz dies sogar noch vor dem Eintreffen der Nachricht erfahren, damit sie nicht durch das Senden der eigenen Nachricht einen irregulären Systemablauf erzeugt. Diese Auswahl zwischen den verschiedenen Möglichkeiten zur Konkatenation ist daher *global*: Die beteiligten Instanzen "wissen", welche Alternative eingegangen werden soll. Diese Problematik wird noch zusätzlich erschwert, wenn in beiden Diagrammen D_2 und D_3 zu Beginn dieselben Nachrichten gesendet werden, bevor die Diagramme unterschiedliches Verhalten zeigen. In diesem Fall muß eine Entscheidung bzgl. der Alternative am Ende von D_1 getroffen werden, obwohl die eigentliche Entscheidung erst bei Auftreten von unterschiedlichem Verhalten von D_2 und D_3 getroffen werden kann (*delayed choice*) [BM95]. Es existieren Ansätze zur Analyse von MSC-Dokumenten im Hinblick auf das Enthalten von nicht-lokaler Auswahl [BAS97].

Durch die Verwendung von vertikaler bzw. alternativer Komposition können auch Schleifen spezifiziert werden, indem am Anfang eines Diagramms dieselbe Bedingung wie am Ende angegeben wird. Da Bedingungen nur als "Klebemarken" interpretiert werden, besitzen sie keine semantische Bedeutung. Somit dienen sie auch nicht zur Synchronisation der Instanzen vor der Ausführung des nächsten Diagramms. Dies kann in Verbindung mit der asynchronen Kommunikation dazu führen, daß eine große Anzahl gesendeter, aber noch nicht empfangener Nachrichten entsteht. Würde z.B. das Diagramm D_1 in Abbildung 5.28 auch zu Beginn die Bedingung C enthalten, könnte das Diagramm iteriert werden. Ist die Instanz I_1 in der Ausführung wesentlich schneller als I_2, kann I_1 eine beliebig große Anzahl von m_1-Nachrichten senden, bevor die erste von ihnen von I_2 empfangen wird. Hierdurch kann der Zustandsraum von Systemen beliebig groß werden, was

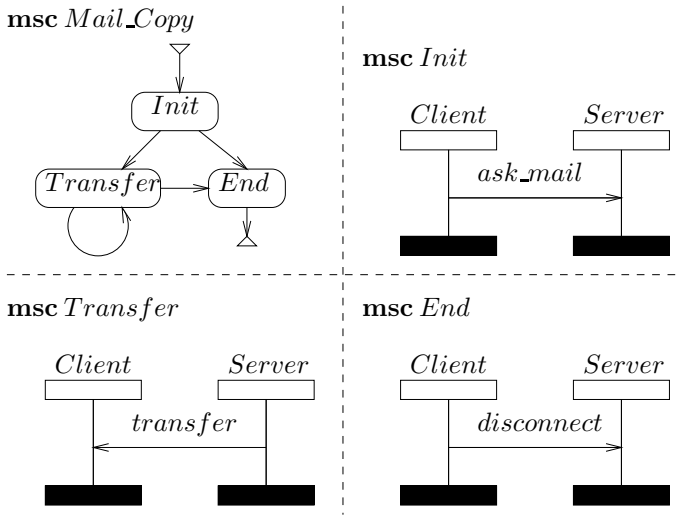


Abbildung 5.29: High-level Message Sequence Chart.

die Anwendungen von Verifikationstechniken wie dem *model checking* [CGP99] erschwert [GHRW98].

Das zweite Verfahren zur Konkatenation von Diagrammen besteht in dem Konzept der *High-level Message Sequence Charts* (HMSC) [MR97]. Hierbei werden Transitionssysteme (siehe Definition 2.1) angegeben, deren Knoten MSCs enthalten. Die Transitionen zwischen den Knoten stellen die kausalen Abhängigkeiten zwischen den MSCs in den Knoten dar. Die Diagramme von Knoten, die nicht durch Transitionen miteinander verbunden sind, sind kausal unabhängig und können nebenläufig ausgeführt werden. Ebenso kann Nebenläufigkeit spezifiziert werden, indem um parallel auszuführende Teil-MSCs ein Rahmen gezeichnet wird [Ren99]. In Abbildung 5.29 ist ein HMSC *Mail_Copy* angegeben, daß den n-Agenten-Diagrammen aus Abbildung 5.22 und Abbildung 5.23 entspricht. Die Knoten *Init*, *Transfer* und *End* repräsentieren die entsprechenden Basisdiagramme. HMSC-Diagramme dürfen verschachtelt werden; daher könnte das HMSC *Mail_Copy* in einem übergeordneten HMSC als Knoten verwendet werden. Auf diese Weise ist eine hierarchische Darstellung des Systemverhaltens möglich.

Im Gegensatz zu den Übersichtsgraphen der n-Agentendiagramme sind die *high-level*-Elemente eine echte Erweiterung der MSCs, da einfache MSCs kein unserem Konkatenationsoperator vergleichbares Kompositionskonzept kennen.

Die bei der Verwendung von Bedingungen für die Konkatenation erläuterten semantischen Probleme existieren auch für HMSC. So ist in Abbildung 5.29 nicht definiert, wie die Entscheidung über eine weitere Iteration des Diagramms *Transfer* getroffen wird, da die Auswahl zwischen *Transfer* und *End* durch eine nicht-lokale Auswahl getroffen wird.

Im Standard MSC 2000 [ITU99] werden High-level Message Sequence Charts

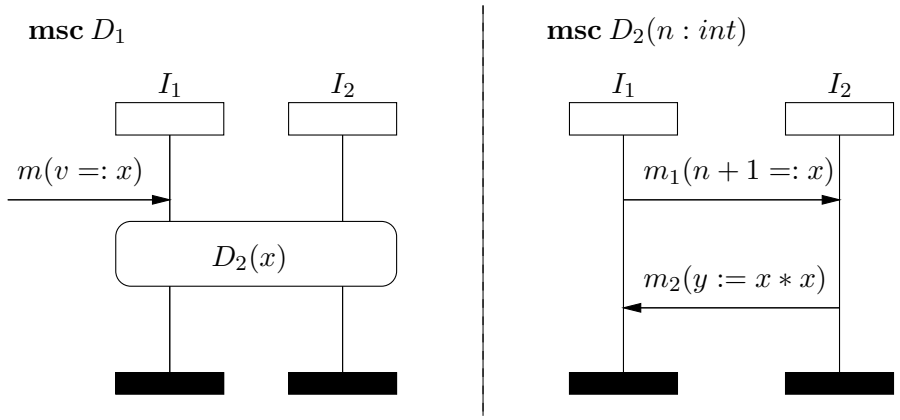


Abbildung 5.30: MSC mit Aufruf eines untergeordneten Diagramms.

um die Behandlung von Daten erweitert. Es wird erlaubt, die Diagramme repräsentierenden Knoten aus HMSCs in Basisdiagramme zu integrieren. In Abbildung 5.30 ist ein Diagramm D_1 angegeben, in dem die Instanz I_1 aus der Umgebung einen Wert v erhält, der an den Bezeichner x gebunden wird. Dann wird die Ausführung mit dem Diagramm D_2 fortgesetzt, wobei der Parameter x übergeben wird. In D_2 wird der Parameter an n gebunden; die Bindung von n ist statisch für die Ausführung von D_2 und kann nicht geändert werden. Nach dem Austausch von Nachrichten endet das von D_2 beschriebene Verhalten und die Ausführung wird mit dem (hier leeren) Rest von D_1 fortgesetzt. Die Frage von Gültigkeitsbereichen wird in [ITU99] nicht diskutiert. So ist zum Beispiel nicht definiert, ob der in D_2 erzeugte Bezeichner y nach Beendigung von D_2 in D_1 weiterhin gültig ist. Ebenso werden nur rudimentäre, informelle Angaben über die Verbindung von Gültigkeitsbereichen und komplexen Sprachkonstrukten wie Alternativen und Schleifen gemacht.

Schleifen und Alternativen. Zur Darstellung komplexen Systemverhaltens beinhalten MSCs das Konzept der *inline expressions* [ITU99, Ren99] (siehe Seite 131). Dabei wird ein Teil eines Diagramms mit einem Rahmen markiert. In der linken oberen Ecke des Rahmens wird durch ein Schlüsselwort die Art des Sprachkonstrukts angegeben. So spezifiziert **alt** eine Auswahl zwischen Alternativen (siehe Abbildung 5.13) und **loop** Schleifen. Ein weiteres Schlüsselwort ist z.B. **opt** für optionales Verhalten. In [ITU99] werden, wie bei der Konkatenation von Diagrammen, wenig Angaben über den Zusammenhang zwischen *inline expressions* und den Gültigkeitsbereichen von Bezeichnern gemacht. So ist z.B. nicht definiert, ob die in einer Alternative definierten Bezeichner nach dem Ende der Alternative weiterhin gültig sind und ob nach einer Auswahl durch unterschiedliche Definitionen in den einzelnen Alternativen offene Terme entstehen könnne.

Im Gegensatz zu den n-Agenten-Diagrammen läßt sich mit MSCs keine Auswahl von Alternativen beschreiben, die sich nur auf eine Instanz erstreckt und trotzdem mit anderen Instanzen interagiert (siehe Abbildung 5.15(a)), da MSCs keine alternativen Eingabeaktionen unterstützen. In diesem Fall muß in allen Alternativen des **alt**-Konstrukts das identische Verhalten der empfangenden Instanz angegeben werden. Zudem entspricht die graphische Aufteilung der Lebenslinie der n-Agenten-Diagramme dem Zeitfluß von oben nach unten, während die alternative Komposition in MSCs entgegen dem Zeitfluß angegeben wird.

Weiterhin muß bei Schleifen und Auswahloperatoren in MSCs nicht festgelegt werden, wie über die Auswahl bzw. eine Fortführung der Schleife entschieden wird. Hierdurch können semantische Mehrdeutigkeiten entstehen. Die Semantik der entsprechenden Sprachkonstrukte der n-Agenten-Diagramme ist hingegen eindeutig, da die Entscheidung einer Kontrollinstanz zugewiesen wird. Hierdurch wird auch festgelegt, welche Bezeichner in der entsprechenden Bedingung auftreten dürfen. Bei der *loop*-Schleife wird zwar die Schleifenkontrolle dezentral vorgenommen; allerdings ist die Schleifenspezifikation nur sinnvoll, wenn in allen Instanzen dieselbe Anzahl von Iterationen durchgeführt wird. Der Benutzer hat durch den vorangehenden Nachrichtenfluß dafür zu sorgen, daß die Anzahl der Iterationen in allen beteiligten Instanzen bekannt ist.

Weitere Konzepte. MSCs beinhalten weitere Konzepte, die wir an dieser Stelle nicht erörtern, da sie über den Sprachumfang der n-Agenten-Diagramme hinausgehen und daher für einen direkten Vergleich der Sprachkonstrukte nicht relevant sind. Zu diesen Konstrukten gehören z.B. die Dekomposition von Instanzen, das Erzeugen neuer Instanzen sowie ein Echtzeitkonzept. Auf die Instanzdekomposition gehen wir im Rahmen möglicher zukünftiger Forschungsarbeiten in Kapitel 7 ein.

Textuelle Notation. Eine textuelle Notation für MSCs ist Bestandteil des MSC-Standards [ITU96a, ITU99]. Sie unterstützt auch die die textuelle Darstellung der High-level MSCs, da diese, im Gegensatz zu den Übersichtsgraphen der n-Agenten-Diagramme, zum Sprachumfang von MSCs gehören und nicht aus den Basisdiagrammen abgeleitet werden können.

Kapitel 6

Semantik der n-Agenten-Diagramme

In diesem Abschnitt definieren wir eine formale Semantik für die n-Agenten-Diagramme aus Kapitel 5. Diese Semantik besteht aus zwei Teilsemantiken: Der Semantik für die Verhaltensbeschreibung des Diagramms und der Semantik für die in den Annotationen angegebenen Datentransformationen. Wir übersetzen den Verhaltensteil in den Kalkül $\mathcal{P}$ aus Kapitel 3, der ebenfalls die Integration von verschiedenen Datensprachen erlaubt. Die Semantik der Datensprache ist von der jeweils gewählten Datensprache abhängig. Als Beispiel verwenden wir die funktionale Sprache aus Kapitel 4.

Die Übersetzung in den Prozeßkalkül bietet den Vorteil, daß die für $\mathcal{P}$ definierten formalen Semantiken (die operationelle und die axiomatische Semantik) aus Kapitel 3 zur Analyse von Spezifikationen verwendet werden können.

Zur Angabe der Semantik definieren wir eine Menge von Funktionen, die das Verhalten der Instanzen eines Sequenzdiagramms in Prozeßdefinitionen aus $\mathcal{P}$ übersetzen. Die Funktionen orientieren sich an den in Abbildung 5.24 in Kapitel 5 angegebenen Produktionen der kontextfreien Grammatik für die textuelle Notation. Zusätzlich wird ein initialer Prozeßterm generiert, der das System initialisiert.

In Abschnitt 6.1 behandeln wir die Übersetzung von Typen und die Darstellung der Tore. In Abschnitt 6.2 wird die Übersetzung von Ausdrücken behandelt, die von der jeweils gewählten Datensprache abhängig ist. Nach der Angabe der Übersetzung von Dokumenten, Diagrammen und Instanzen in Abschnitt 6.3 folgt in Abschnitt 6.4 die Semantik der einzelnen Ereignisse auf den Lebenslinien der Instanzen. Abschnitt 6.5 enthält als Beispiel die Semantik für die Spezifikation des Gefangenendilemmas aus Abschnitt 5.3.5. In Abschnitt 6.6 wird die Konkatenation von Diagrammen behandelt und anhand eines Client-Server-Beispiels erläutert. Nach einer Beschreibung der Einschränkungen der Semantik bzgl. verschachtelter *loop*-Schleifen und *race conditions* in Abschnitt 6.7 werden in Abschnitt 6.8 verwandte Semantiken für graphische Notationen diskutiert.

6.1 Typsystem und Tore

Analog zu den Anforderungen an ein Typsystem DTYPE für eine Datenteilsprache in Abschnitt 3.1 definieren wir Anforderungen an ein allgemeines Typsystem GTYPE für die Datensprache, das die Typisierung von Toren unterstützt.

Definition 6.1 *Sei* GTYPE *eine Menge von Datentypen* T, T'*, wobei* GTYPE *die folgenden Anforderungen erfüllt:*

- *bool* $\in$ GTYPE, *int* $\in$ GTYPE.
- *Für* $T_1 \in$ GTYPE, ..., $T_n \in$ GTYPE *gilt auch* $T_1 \times ... \times T_n \in$ GTYPE.
- *Für* $T \in$ GTYPE *gilt auch* T gate $\in$ GTYPE.
- *Für* $T, T' \in$ GTYPE *gilt auch* T returns $T' \in$ GTYPE.

Das Typsystem GTYPE muß als Basistypen mindestens Wahrheitswerte und ganze Zahlen enthalten. Weiterhin muß die Verwendung von Tupeln möglich sein, da dieser Datentyp für die Realisierung von Anfragen notwendig ist. Zu diesen Typen müssen auch die zugehörigen Operatoren und Zugriffsfunktionen definiert sein (arithmetische und logische Operatoren sowie eine Indexfunktion für den Zugriff auf Tupelelemente). Die Typausdrücke T *gate* geben den Typ von Nachrichten ohne Antwort an, die Ausdrücke T_1 *returns* T_2 repräsentieren die Typen von Anfragen. Wie der Typkonstruktor *channel* in Kapitel 3 können die Konstruktoren *gate* und *returns* beliebig tief verschachtelt werden.

Definition 6.2 *Sei* GATES *die Menge aller Tornamen, sei* DIAG *die Menge aller Diagramme, sei* INST *die Menge aller Instanzen. Die Systemumgebung bezeichnen wir mit dem speziellen Namen* **env** $\in$ INST*. Die Tornamen einer Instanz* $I \in$ INST *bezeichnen wir mit gates*(I)*. Wir nehmen an, daß jede Instanz* $I \in$ INST $\setminus \{$**env**$\}$ *die beiden implizit deklarierten Tore* $\_cont_I$: *int gate und* $\_while_I$: *bool gate enthält. Die Funktion type_of*$(.,.)$: GATES $\times$ INST $\rightarrow$ GTYPE *liefert den Typ eines Tornames in einer Instanz.*

Da verschiedene Instanzen Tore gleichen Namens enthalten können, liefert die Funktion *type_of*(m, I) den Typ des Tornamens m in Instanz I. Wir zählen die Umgebung **env** zu den Instanzen, da auf diese Weise die Definition von Funktionen wie *gates* vereinfacht wird. Die Tore $\_while_I$ und $\_cont_I$ für $I \in$ INST $\setminus \{$**env**$\}$ dienen zur Steuerung von Schleifen und Diagrammkonkatenation.

Wir stellen die Tore der n-Agenten-Diagramme durch die Kanäle der Datensprache dar. Entsprechend werden die Nachrichten an ein Tor durch Kommunikation über den entsprechenden Kanal realisiert. Jedes Tor wird hierbei durch einen eindeutigen Kanal repräsentiert, um die korrekte Adressierung der Instanzen zu gewährleisten.

Der Kalkül $\mathcal{P}$ setzt ein Typsystem DTYPE der verwendeten Datensprache voraus. Daher definieren wir eine Version von DTYPE, die aus GTYPE abgeleitet ist. Hierbei müssen die Konstruktoren T *gate* und T_1 *returns* T_2 aus GTYPE in entsprechende Typausdrücke in DTYPE umgewandelt werden. Wir stellen Tore durch die Kanäle der Datensprache dar, daher wandeln wir die Typen T *gate* und T_1 *returns* T_2 aus GTYPE in Kanaltypen aus DTYPE um.

Definition 6.3 *Wir definieren das zu* GTYPE *gehörende Typsystem* DTYPE *wie folgt:*

- *Für jedes* $T \in$ GTYPE*, in dem keine Teilausdrücke* T' *gate oder* T' *returns* T'' *für Tore vorkommen, gilt* $T \in$ DTYPE*.*
- *Für jedes* $T \in$ DTYPE *gilt auch* T *channel* $\in$ DTYPE*.*

DTYPE enthält also alle Typausdrücke aus GTYPE, die nicht die Typen von Toren darstellen; die Typen für Tore werden durch Typausdrücke für Kanäle ersetzt.

Zur Übersetzung von Typausdrücken von GTYPE nach DTYPE definieren wir die folgende Funktion *typeconv*.

Definition 6.4 *Wir definieren die Funktion typeconv* : GTYPE $\rightarrow$ DTYPE *wie folgt:*

- $typeconv(T) = T$ *für Basistypen* T
- $typeconv(T\ c) = typeconv(T)\ c$ *für Typkonstruktoren* c *mit* $c \neq gate$
- $typeconv(T\ gate) = typeconv(T)\ channel$
- $typeconv(T_1\ returns\ T_2) = (T'_{11} \times ... \times T'_{1n} \times T'_2\ channel)\ channel$
  für $typeconv(T_1) = (T'_{11} \times ... \times T'_{1n})$ *und* $typeconv(T_2) = T'_2$

In der vierten Regel interpretieren wir den Fall $T = T_1$ als $T = (T_1)$.

Für jedes Tor m einer Instanz I nehmen wir die Existenz eines Kanals $m_{\$T'\$}$ an, wobei m vom Typ T ist und $T' = typeconv(T)$ gilt. Für ein Tor des Typs T *gate* hat der zugehörige Kanal den Typ $typeconv(T\ gate) = T'$ *channel*, wobei $T' = typeconv(T)$ die Übersetzung von T ist. Handelt es sich um ein Tor für Anfragen, d.h. ein Tor mit Wertrückgabe, überträgt der Kanal zusätzlich einen Kanalwert, auf dem der Rückgabewert an den Aufrufer gesendet wird. Hat ein Tor den Typ T_1 *returns* T_2, hat der zugehörige Kanal den Typ $(T'_1 \times T'_2\ channel)\ channel$. Ist T_1 selbst ein Tupel $T_{11} \times ... \times T_{1n}$, erweitern wir diesen Tupel um eine Komponente, so daß der Kanal den Typ $(T'_{11} \times ... \times T'_{1n} \times T'_2\ channel)\ channel$ besitzt. Durch diese Vereinfachung ermöglichen wir einen erleichterten Zugriff auf die einzelnen Komponenten des Tupels, da wir auf die

bei hierarchischen Tupeln notwendige mehrfache Anwendung von Komponentenfunktionen verzichten können.

Tornamen können in verschiedenen Instanzen für die Benennung von Toren verwendet werden. Da jedes Tor durch einen eindeutigen Kanal dargestellt wird, definieren wir die folgenden Funktionen, die die Beziehungen zwischen Toren und den zugehörigen Kanälen angeben.

Definition 6.5 *Für ein Tor $m \in$ GATES, das in der Instanz $I \in$ INST definiert wurde, bezeichnet $chan(m, I) \in$ CHAN den zugehörigen Kanal. Die inverse Funktion $gate(c)$ liefert zu einem Kanalwert c den originären Tornamen und die zugehörige Instanz I. Es gilt $chan(m, I) = c$ gdw. $gate(c) = (m, I)$ für $m \in$ GATES und $c \in$ CHAN.*

Die Funktion $name(.)$ liefert den Namen eines Kanals ohne seine Typangabe: $name(c_{\$T\$}) = c$.

Wir benötigen die Funktion *name*(.) zur Generierung des Bezeichners, an den ein Kanal gebunden wird. Um die Lesbarkeit der erzeugten Prozeßdefinitionen zu erhöhen, binden wir einen Kanal $c_{\$T\$}$ an den Bezeichner $name(c_{\$T\$}) = c$.

6.2 Ausdrücke

Die jeweils in die n-Agenten-Diagramme integrierte Datensprache muß über eine Reduktionssemantik verfügen, die die Reduktion der Ausdrücke formal definiert. Die Semantik der n-Agenten-Diagramme wird dann, wie die Semantik des Kalküls $\mathcal{P}$ aus Kapitel 3, mit dieser Reduktionssemantik parametrisiert. Hierzu müssen eventuell Anpassungen bzgl. der Datentypen aus GTYPE vorgenommen werden. Eventuell ist es auch möglich, komplexe Datentypen in den n-Agenten-Diagrammen durch einfachere Datentypen in der Semantik nachzubilden. Daher geben wir für die Ausdrücke Übersetzungsfunktionen an. Für die Übersetzung der Ausdrücke nehmen wir eine Funktion $[\![.]\!]^{\mathcal{E}}_{expr}$ an, die die Ausdrücke $E \in L(expr)$ in semantisch äquivalente Ausdrücke der Datensprache übersetzt. Diese Funktion ist von der gewählten Datensprache abhängig.

Als Beispiel geben wir in Abbildung 6.1 eine Übersetzung für die Datensprache aus Kapitel 4 an, da für diese Sprache eine Reduktionssemantik vorliegt. Für diese Sprache muß keine Übersetzung im eigentlichen Sinn vorgenommen werden. Da aber auch andere Datensprachen in die Diagramme integriert werden sollen, geben wir eine Übersetzungsfunktion an, die als Beispiel für andere Übersetzungen fungiert.

In Abbildung 6.1 werden bei der Übersetzung von λ-Ausdrücken die Typausdrücke konvertiert. Dies wird durch Anwendung der Funktion *typeconv* realisiert. Da λ-Abstraktionen beliebig in Ausdrücken auftreten können, muß die Übersetzung rekursiv auf alle Teilausdrücke eines Ausdrucks angewendet werden. Daher ist die Funktion in Abbildung 6.1 induktiv definiert. Da Tornamen nicht in

Ausdrücken der Datensprache auftreten dürfen, ist, wie zuvor erwähnt, für die Sprache aus Kapitel 4 eine Konvertierung der Typausdrücke in λ-Abstraktionen eigentlich nicht notwendig. Wir geben die Konvertierung trotzdem an, da die Funktion $[\![.]\!]^{\mathcal{E}}_{expr}$ als Beispiel für eine allgemeine Konvertierung dienen soll und bei der Übersetzung von anderen Datensprachen die Konvertierung weiterer Datentypen erforderlich sein könnte.

Zusätzlich muß bei der Übersetzung eines *let* $x = E_1$ *in* E_2 oder einer λ-Abstraktion der Bezeichner x in die Menge der bekannten Bezeichner $\mathcal{E}$ bei der Übersetzung von E_2 eingefügt werden. Die Menge $\mathcal{E}$ ist eine Menge von Bezeichnern, die zum Zeitpunkt der Übersetzung des Ausdrucks gültig sind. Sie wird z.B. in der Semantik bei der Übersetzung von Schleifen benötigt. Da Bezeichner, die in einem Ausdruck definiert werden, nur innerhalb dieses Ausdrucks gültig sind, ist auch hier eine Modifikation von $\mathcal{E}$ bei der Übersetzung der Datensprache aus Kapitel 4 nicht notwendig. Wir führen die Modifikation trotzdem durch, da Übersetzungen für andere Sprachen die Informationen in $\mathcal{E}$ eventuell benötigen (z.B. bei Einführung eines imperativen Datentyps wie den Referenztypen aus SML [HMM86, Pau91]).

Die Funktion $[\![E_1, ..., E_n]\!]^{\mathcal{E}}_{expr\_list}$ dient zur Übersetzung einer Liste von Ausdrücken und wird zur Behandlung der Parameter in Nachrichten verwendet. Eine gesonderte Behandlung der Tornamen ist bei der Übersetzung von Ausdrücken nicht notwendig, da Tornamen nicht in Datenausdrücken auftreten dürfen.

6.3 Dokumente, Diagramme und Instanzen

Zur Darstellung eines Sequenzdiagramms mit Daten in der Prozeßalgebra $\mathcal{P}$ erzeugen wir eine Prozeßumgebung Θ (siehe Seite 35), die die Semantik der einzelnen Instanzen in Form von Prozeßdefinitionen enthält. Weiterhin generieren wir einen Term, der die initiale Systemkonfiguration erzeugt.

Zur Übersetzung von Instanzen benötigen wir die folgenden Hilfsfunktionen zur Erzeugung von Parameterlisten:

Definition 6.6 *Sei make_vect* : $2^{\text{VAR}} \rightarrow \text{VAR}^*$ *eine Funktion, die aus einer Menge von Bezeichnern mit n Elementen einen Vektor der Länge n erzeugt, wobei die Elemente lexikographisch geordnet werden [Ern92]. Sei to_set* : $\text{VAR}^* \rightarrow 2^{\text{VAR}}$ *die inverse Funktion. Analog sei chan_vect* : $2^{\text{CHAN}} \rightarrow \text{CHAN}^*$ *eine Funktion, die eine Menge von Kanalwerten lexikographisch zu einem Vektor sortiert.*

Definition 6.7 *Sei init eine Funktion, die das initiale Diagramm* $d \in \text{DIAG}$ *eines Dokuments liefert.*

Besteht das Dokument nur aus einem Diagramm, liefert *init* den Namen dieses Diagramms.

$$[\![v]\!]^{\mathcal{E}}_{expr} = v$$
$$[\![x]\!]^{\mathcal{E}}_{expr} = x$$
$$[\![(E_1, ..., E_n)]\!]^{\mathcal{E}}_{expr} = ([\![E_1]\!]^{\mathcal{E}}_{expr}, ..., [\![E_n]\!]^{\mathcal{E}}_{expr})$$
$$[\![[E_1, ..., E_n]]\!]^{\mathcal{E}}_{expr} = [[\![E_1]\!]^{\mathcal{E}}_{expr}, ..., [\![E_n]\!]^{\mathcal{E}}_{expr}]$$
$$[\![op_{un} E]\!]^{\mathcal{E}}_{expr} = op_{un}\ [\![E]\!]^{\mathcal{E}}_{expr}$$
$$[\![E_1\, op_{bin} E_2]\!]^{\mathcal{E}}_{expr} = [\![E_1]\!]^{\mathcal{E}}_{expr}\ op_{bin}\ [\![E_2]\!]^{\mathcal{E}}_{expr}$$
$$[\![\mathit{if}\ \ E_1\ \mathit{then}\ \ E_2\ \mathit{else}\ \ E_3]\!]^{\mathcal{E}}_{expr} = \mathit{if}\ \ [\![E_1]\!]^{\mathcal{E}}_{expr}\ \mathit{then}\ \ [\![E_2]\!]^{\mathcal{E}}_{expr}\ \mathit{else}\ \ [\![E_3]\!]^{\mathcal{E}}_{expr}$$
$$[\![\text{let}\ \ x = E_1\ \ \text{in}\ \ E_2]\!]^{\mathcal{E}}_{expr} = \text{let}\ \ x = [\![E_1]\!]^{\mathcal{E}}_{expr}\ \ \text{in}\ \ [\![E_2]\!]^{\mathcal{E}\cup\{x\}}_{expr}$$
$$[\![\lambda x : T.E]\!]^{\mathcal{E}}_{expr} = \lambda x : typeconv(T).[\![E]\!]^{\mathcal{E}\cup\{x\}}_{expr}$$

$$[\![E_1, ..., E_n]\!]^{\mathcal{E}}_{expr\_list} = [\![E_1]\!]^{\mathcal{E}}_{expr}, ..., [\![E_n]\!]^{\mathcal{E}}_{expr}$$

Abbildung 6.1: Übersetzungsfunktionen für Ausdrücke.

Definition 6.8 *Die Funktion initpar* : DIAG $\times$ (INST $\setminus$ {**env**}) $\rightarrow$ (VAR $\times$ GTYPE)* *liefert die initialen Datenparameter und die zugehörigen Typen der Instanz* $I \in$ INST $\setminus$ {**env**} *in Diagramm* $d \in$ DIAG.

Die Funktion $initpar(D, I)$ berechnet die Parameter der Instanz I in Diagramm D, die durch die Parameterübergabe bei der Konkatenation von Diagrammen mit Werten belegt werden.

Zur Übersetzung von Diagrammen geben wir eine Menge von Funktionen an, die die einzelnen Bestandteile der Diagramme übersetzen. Diese Funktionen orientieren sich an den Regeln der kontextfreien Grammatik in Abbildung 5.24.

6.3.1 Dokumente.

Die Übersetzungsfunktionen für Diagramme und Instanzen sind in Abbildung 6.2 angegeben. Diese Funktionen erzeugen zum einen eine Prozeßumgebung Θ (siehe Seite 35), deren Prozeßdefinitionen die Semantik der einzelnen Instanzen in den verschiedenen Diagrammen darstellen. Dabei wird für jede Instanz in jedem Diagramm eine eigene Prozeßdefinition erzeugt. Außerdem werden bei der Verwendung von Schleifen Hilfsprozeßdefinitionen in Θ eingefügt, die zur Darstellung der Semantik des Schleifenverhaltens benötigt werden.

$$[\![diag_1 \ldots diag_l]\!]_{document} =$$
$$\quad \text{let } \_ = [\![diag_1 \ldots diag_l]\!]_{diagrams} \text{ in}$$
$$\quad \textbf{ch } channels.\ spawn(D\_I_1\_start(params)); \ldots; spawn(D\_I_n\_start(params))$$

mit $D = init()$,
$\textsc{Inst} = \{I_1, \ldots, I_n, \textbf{env}\}$,
$instgates = gates(I_1) \uplus \ldots \uplus gates(I_n)$
$allgates = instgates \uplus gates(\textbf{env}) = \{m_1, \ldots m_k\}$,
$channels = \{chan(m_i, I) \mid m_i \in instgates, m_i \in gates(I)\}$
$params = chan\_vect(channels)$

$$[\![diag_1 \ldots diag_l]\!]_{diagrams} =$$
$$\quad \text{let } \_ = [\![diag_1]\!]_{diagram} \text{ in } \ldots \text{ in } [\![diag_n]\!]_{diagram}$$
$$[[\textbf{initial}]\ \textbf{diagram}\ D\ inst\_list]\!]_{diagram} = [\![instance\_list]\!]^D_{instances}$$
$$[\![inst_1 \ldots inst_n]\!]^D_{instances} =$$
$$\quad \text{let } \_ = [\![inst_1]\!]^D_{instance} \text{ in } \ldots \text{ in } [\![inst_n]\!]^D_{instance}$$

$$[\![\textbf{instance } I\ [\ \textbf{uses } param\_list\ ]\ [\ \textbf{gates } gatelist\ ]\ \textbf{events } events]\!]^D_{instance} =$$
$$\quad \textit{if } D = init() \textit{ then}$$
$$\quad\quad include\_\Theta(D\_I\_start(vect) \mapsto [\![events]\!]^{D,I,env}_{events})$$
$$\quad \textit{else}$$
$$\quad\quad include\_\Theta(D\_I\_continue(vect) \mapsto [\![events]\!]^{D,I,env}_{events})$$

mit $initparams = initpar(D, I) = x_1 : T_1, \ldots, x_j : T_j$
$channels = \{c_1, \ldots, c_k\}$
$gate\_vect = chan\_vect(channels) = c_{i1}, \ldots, c_{ik}$
$vect = name(c_{i1}) : typeconv(type\_of(gate(c_{i1}, I))),$
$\quad \ldots,$
$\quad name(c_{ik}) : typeconv(type\_of(gate(c_{ik}, I))),$
$\quad x_1 : typeconv(T_1), \ldots, x_j : typeconv(T_j)$
$env = \{name(c_{i1}), \ldots, name(c_{ik})\} \cup \{x_1, \ldots, x_j\}$

Abbildung 6.2: Übersetzungsfunktionen für Instanzen.

Weiterhin wird ein initialer Prozeßterm generiert, der zur Initialisierung des Systems dient und die Prozeßdefinitionen aus Θ verwendet. In diesem Term werden die Kanäle erzeugt, die die initialen Tore der Instanzen realisieren. In Abbildung 6.2 nehmen wir an, daß das zu übersetzende Diagramm die Instanzen $I_1, ..., I_n$ enthält. Wir bilden die Menge *instgates* aus der disjunkten Vereinigung der Tore der Instanzen[1]. Die Menge *channels* enthält die den Toren aus *instgates* durch die Funktion *chan*(.,.) zugeordneten Kanäle. Diese Kanäle werden anfänglich durch den Operator für Kanalerzeugung restringiert, um die Kommunikation zwischen den Instanzen zu erzwingen. Die Menge *allgates* erhält zusätzlich die Tore der Umgebung **env**. Diese zusätzlichen Kanäle werden nicht restringiert, damit eine Interaktion mit der Umgebung möglich ist. Die Menge *channels* enthält dann die zugehörigen Kanäle für die Tore aus *allgates*. Diese Kanäle werden mit der Funktion *chan_vect* lexikographisch zu einem Vektor *params* geordnet. Die Aufrufe der D_I_start-Prozesse mit $I \in \textsc{Inst} \setminus \{\mathbf{env}\}$ erhalten dann diesen Kanalvektor, so daß die Kanäle in allen Instanzen bekannt sind. Die Übergabe der Kanäle in den Parameterlisten der Prozeßaufrufe ist notwendig, da in Kapitel 3 gefordert wird, daß eine Prozeßdeklaration keine freien Kanäle enthalten darf (siehe die Forderung $fc(t) = \emptyset$ für Deklarationen $P(\vec{x}) \mapsto t$ auf Seite 36).

Die Übersetzung eines Dokuments beginnt mit der Funktion $[\![.]\!]_{document}$. Als Ergebnis liefert diese Funktion den initialen Prozeßterm. Gleichzeitig werden bei der Übersetzung der einzelnen Diagramme eines Dokuments durch die Funktion $[\![.]\!]_{diagrams}$ Prozeßdefinitionen erzeugt, die das Verhalten der einzelnen Instanzen während des Systemablaufs realisieren. Wir nehmen an, daß das Einfügen von Prozeßdefinitionen in Θ durch eine Funktion $include\_\Theta : \textsc{Proc} \rightarrow bool$ mit Hilfe von Seiteneffekten erfolgt. Da Prozeßdefinitionen an verschiedenen Stellen des Übersetzungsvorgangs erzeugt werden, erlaubt die Verwendung einer Funktion mit Seiteneffekten eine kompaktere Angabe der Übersetzungsfunktionen; andernfalls müßte Θ als weiterer Parameter der Funktionen realisiert werden. Als Rückgabewert gibt $include\_\Theta$ den booleschen Wert *true* zurück[2]. Die Prozeßdefinitionen für die Instanzen werden in den $[\![.]\!]_{diagrams}$ untergeordneten Funktionen durch die Funktion $include\_\Theta$ durch Seiteneffekte in die Prozeßumgebung Θ eingefügt.

Jeder Instanz I im Diagramm D wird von $[\![.]\!]_{instance}$ eine Prozeßdefinition D_I_start bzw. $D\_I\_continue$ zugeordnet. Mit D_I_start beginnt die Ausführung der Instanz im initialen Diagramm; mit $D\_I\_continue$ wird sie nach einer Konkatenation fortgesetzt.

[1] Hierzu gehören durch die Definition von *gates*(.) in Definition 6.2 auch die impliziten Tore $\_cont_I, \_while_I$ für $I \in \{I_1, ..., I_n\}$.

[2] Da der Rückgabewert bei der Übersetzung nicht verwendet wird, könnte auch jeder andere beliebige Wert zurückgegeben werden.

6.3.2 Diagramme.

Die einzelnen Diagramme werden mit $[\![.]\!]_{diagram}$ übersetzt. Jeder Aufruf von $[\![.]\!]_{diagram}$ erzeugt einen Aufruf der Funktion $[\![.]\!]^D_{instances}$, mit der die einzelnen Instanzen eines Diagramms übersetzt werden. Diese Funktion erhält als Index den Namen d des zu übersetzenden Diagramms. $[\![.]\!]^D_{instances}$ wiederum verwendet die Funktion $[\![.]\!]^D_{instance}$ zum Übersetzen der einzelnen Instanzen.

6.3.3 Instanzen.

Die Funktion $[\![.]\!]^D_{instance}$ übersetzt innerhalb eines Diagramms D eine einzelne Instanz I und erzeugt hierfür eine Prozeßdefinition. Handelt es sich um das initiale Diagramm, wird eine Prozeßdefinition D_I_start generiert, andernfalls heißt die Definition $D\_I\_continue$. Diese Unterscheidung erhöht die Lesbarkeit der Übersetzung. Die Prozeßdefinition erwartet als Parameter zum einen den lexikographisch geordneten Vektor der global bekannten Kanäle. Diese Kanäle werden an gleichnamige Bezeichner gebunden, bei denen die Typangabe durch Anwendung der Funktion $name(.)$ weggelassen wird. Neben den Bezeichern werden in der Parameterliste auch die Typen der einzelnen Bezeichner angegeben, welche zuvor mit den Funktionen $typeconv$, $type\_of$ und $gate$ ermittelt und dann in das Typsystem DTYPE umgewandelt worden sind. Zum zweiten enthält die Parameterliste die Aufrufparameter $x_1, ..., x_j$, die bei der Konkatenation von Diagrammen übergeben werden. Wir ordnen diese Aufrufparameter hinter dem Vektor der Kanalparameter an. Der Vektor der Aufrufparameter und ihrer Typen wird von der Funktion $initpar$ berechnet. In diesem Vektor werden auch die Kanäle für dynamisch mit dem **gate**-Konstrukt erzeugte Tore übergeben.

Die Sequenz der Ereignisse auf der Lebenslinie von I wird mit der Funktion $[\![.]\!]^{D,I,env}_{events}$ übersetzt. Die Menge env der bekannten Bezeichner enthält dabei die Bezeichner für die Kanäle sowie die Aufrufparameter der Instanz. Zusätzlich wird neben dem Diagrammnamen D der Name I der jeweils zu übersetzenden Instanz angegeben, da diese Information in den untergeordneten Übersetzungsfunktionen zur korrekten Übersetzung von Schleifen benötigt wird (siehe Abbildung 6.10).

Beispiel 6.9 (Fortsetzung des Gefangenendilemma-Beispiels)
Im folgenden stellen wir die Konfiguration für den Systemstart des Gefangenendilemmas aus Abschnitt 5.3.5 und Beispiel 5.6 dar.

$$\begin{aligned}
&\textbf{ch}\ \{gegner_{A\$bool\$}, gegner_{B\$bool\$}, runden_{A\$int\$}, runden_{B\$int\$},\\
&\qquad wahl_{A\$bool\$}, wahl_{B\$bool\$}, _cont_{A\$int\$}, _cont_{B\$int\$}, _cont_{S\$int\$},\\
&\qquad _while_{A\$bool\$}, _while_{B\$bool\$}, _while_{C\$bool\$}, \}.\\
&spawn(Gefangenendilemma_Spieler_A_start(\\
&\qquad ergebnis_{\$int\times int\$}, gegner_{A\$bool\$}, gegner_{B\$bool\$}, runden_{A\$int\$},
\end{aligned}$$

$$
\begin{array}{l}
\quad runden_{B\$int\$}, runden_{S\$int\$}, wahl_{A\$bool\$}, wahl_{B\$bool\$}, \\
\quad \_cont_{A\$int\$}, \_cont_{B\$int\$}, \_cont_{S\$int\$}, \\
\quad \_while_{A\$bool\$}, \_while_{B\$bool\$}, \_while_{S\$bool\$})); \\
spawn(Gefangenendilemma\_Schiedsrichter\_start(\\
\quad ergebnis_{\$int\times int\$}, gegner_{A\$bool\$}, gegner_{B\$bool\$}, runden_{A\$int\$}, \\
\quad runden_{B\$int\$}, runden_{S\$int\$}, wahl_{A\$bool\$}, wahl_{B\$bool\$}, \\
\quad \_cont_{A\$int\$}, \_cont_{B\$int\$}, \_cont_{S\$int\$}, \\
\quad \_while_{A\$bool\$}, \_while_{B\$bool\$}, \_while_{S\$bool\$})); \\
spawn(Gefangenendilemma\_Spieler\_B\_start(\\
\quad ergebnis_{\$int\times int\$}, gegner_{A\$bool\$}, gegner_{B\$bool\$}, runden_{A\$int\$}, \\
\quad runden_{B\$int\$}, runden_{S\$int\$}, wahl_{A\$bool\$}, wahl_{B\$bool\$}, \\
\quad \_cont_{A\$int\$}, \_cont_{B\$int\$}, \_cont_{S\$int\$}, \\
\quad \_while_{A\$bool\$}, \_while_{B\$bool\$}, \_while_{S\$bool\$})); \\
\text{mit } \Theta\text{:} \\
Gefangenendilemma\_Spieler\_A\_start(vect) \mapsto [\![events_A]\!]^{D,A,\mathcal{E}}_{events} \\
Gefangenendilemma\_Spieler\_B\_start(vect) \mapsto [\![events_B]\!]^{D,B,\mathcal{E}}_{events} \\
Gefangenendilemma\_Schiedsrichter\_start(vect) \mapsto [\![events_S]\!]^{D,S,\mathcal{E}}_{events} \\
\\
D = Gefangenendilemma, \\
\mathcal{E} = \\
\quad \{ergebnis, gegner_A, gegner_B, runden_A, runden_B, runden_S, wahl_A, wahl_B, \\
\quad \_cont_A, \_cont_B, \_cont_S, \_while_A, \_while_B, \_while_S\} \\
\\
\text{und } vect = \\
\quad ergebnis : (int \times int)\ channel,\ gegner_A : bool\ channel, \\
\quad gegner_B : bool\ channel,\ runden_A : int\ channel,\ runden_B : int\ channel, \\
\quad runden_S : int\ channel,\ wahl_A : bool\ channel,\ wahl_B : bool\ channel, \\
\quad \_cont_A : int\ channel, \_cont_B : int\ channel, \_cont_S : int\ channel, \\
\quad \_while_A : bool\ channel, \_while_B : bool\ channel, \_while_S : bool\ channel
\end{array}
$$

Die Kanäle für die Tore der Instanzen und der Umgebung sind als globale Konstanten beim Systemstart bekannt. Wir verwenden die Abkürzungen A, B und S für Spieler_A, Spieler_B bzw. Schiedsrichter. Alle Kanäle außer $runden_{S\$int\$}$ *und* $ergebnis_{\$int\times int\$}$ *werden durch einen* **ch***-Operator restringiert, so daß die Kommunikation zwischen den Systemkomponenten erzwungen wird; die beiden übrigen Kanäle dienen zur Interaktion mit der Umgebung, daher darf ihre Gültigkeit nicht auf den initialen Prozeßterm beschränkt sein.*

Die Prozeßumgebung Θ *enthält die Prozeßdefinitionen* $Gefangenendilemma\_Spieler\_A\_Start$, $Gefangenendilemma\_Spieler\_B\_Start$ *und* $Gefangenendilemma\_Schiedsrichter\_Start$, *die die Übersetzung der Lebenslinie der jeweiligen Instanz beinhalten. Jede Prozeßdefinition erwartet als Parameter die Kanäle, die*

die anfänglich bekannten Toren realisieren. Hierzu kommen auch die impliziten Tore $\_cont_I$ *bzw.* $\_while_I$ *zur Steuerung von Schleifen und Konkatenation. Da diese Parameter bei allen Prozessen übereinstimmen, geben wir sie durch den Vektor vect an; die Reihenfolge der Parameter in vect beruht auf der lexikographischen Ordnung. Da kein Torname in mehr als einer Instanz auftritt, kann die Funktion chan die Tornamen direkt auf gleichnamige Kanalwerte abbilden. Daher entsprechen die Bezeichner in vect den Tornamen aus allgates.*

6.4 Ereignisse

Die Sequenz von Ereignissen, aus denen die Lebenslinie einer Instanz zusammengesetzt ist, wird mit der Funktion $[\![.]\!]^{D,I,\mathcal{E}}_{events}$ übersetzt, die im unteren Teil von Abbildung 6.3 angegeben ist. Handelt es sich um ein einzelnes Ereignis, wird dieses direkt mit der Funktion $[\![.]\!]^{D,I,\mathcal{E}}_{event}$, die im oberen Teil der Abbildung angegeben, ist, übersetzt. Andernfalls wird das erste Ereignis der Sequenz übersetzt. Die Übersetzung der restlichen Sequenz geschieht durch einen rekursiven Aufruf $[\![.]\!]^{D,I,\mathcal{E}'}_{events}$, wobei die neue Menge $\mathcal{E}'$ neben den bisher bekannten Bezeichnern aus $\mathcal{E}$ auch die vom ersten Ereignis der Sequenz definierten Bezeichner enthält.

Zur Berechnung der neu definierten Bezeichner wird die Funktion $bound\_by : L(events) \times \textsc{Inst} \rightarrow 2^{\textsc{Var}}$ aus Abbildung 6.4 verwendet. Hierbei wird neben dem Ereignis auch die Instanz übergeben, um die Menge der neuen Bezeichner eines **gate**-Konstrukts berechnen zu können. Der obere Teil der Abbildung enthält die Definition von *bound_by* für Kommunikationsereignisse; hier werden nur bei einem Eingabeereignis mit der **bind**-Option neue Bezeichner eingeführt. Die übrigen syntaktischen Konstrukte werden im unteren Teil der Abbildung behandelt. Bei der dynamischen Torerzeugung mit **gate** werden die Namen der Bezeichner, an die die neu erzeugten zugehörigen Kanäle gebunden wurden, als Ergebnis geliefert. Bei einer Auswahl mit **select** werden keine neuen Bezeichner eingeführt, die nach der Auswahl noch gültig sind, da sich die Gültigkeit von Bezeichnern in einer Alternative nur auf diese Alternative beschränkt. Die im Rumpf einer Schleife definierten Bezeichner sind hingegen auch in den Aktionen nach dem Schleifenende bekannt, so daß die Funktion die im Schleifenrumpf definierten Bezeichner liefert.

Zur Übersetzung einer Sequenz von Ereignissen, bei der das erste Element eine Schleife ist, verwendet $[\![.]\!]^{D,I,\mathcal{E}}_{events}$ die Funktionen $[\![.]\!]^{D,I,\mathcal{E}}_{loop}$ bzw. $[\![.]\!]^{D,I,\mathcal{E}}_{while}$, da die Übersetzung von sequentieller Komposition mit Schleifen gesondert behandelt werden muß.

Die Funktion $[\![.]\!]^{D,I,\mathcal{E}}_{event}$ übersetzt einzelne Ereignisse. Im folgenden behandeln wir die Übersetzungen der einzelnen Typen von Ereignissen.

$$[\![\textbf{skip}]\!]^{D,I,\mathcal{E}}_{event} = \mathbf{1}$$

$$[\![comm\_event]\!]^{D,I,\mathcal{E}}_{event} = [\![comm\_event]\!]^{D,I,\mathcal{E}}_{comm\_event}$$

$$[\![\textbf{choose } ev \ ...]\!]^{D,I,\mathcal{E}}_{event} = [\![ev]\!]^{D,I,\mathcal{E}}_{comm\_event}$$

$$[\![data\_decl]\!]^{D,I,\mathcal{E}}_{event} = [\![data\_decl]\!]^{D,I,\mathcal{E}}_{data}$$

$$[\![chan\_decl]\!]^{D,I,\mathcal{E}}_{event} = [\![chan\_decl]\!]^{D,I,\mathcal{E}}_{gate}$$

$$[\![choice]\!]^{D,I,\mathcal{E}}_{event} = [\![choice]\!]^{D,I,\mathcal{E}}_{choice}$$

$$[\![\textbf{loop } ident \textbf{ times instances } id\_list \textbf{ do } events \textbf{ end loop}]\!]^{D,I,\mathcal{E}}_{event} = [\![\textbf{loop } ident \textbf{ times instances } id\_list \textbf{ do } events \textbf{ end loop; skip}]\!]^{D,I,\mathcal{E}}_{loop}$$

$$[\![\textbf{while } E \textbf{ in } I \textbf{ instances } id\_list \textbf{ do } events_1 \textbf{ end while}]\!]^{D,I,\mathcal{E}}_{event} = [\![\textbf{while } E \textbf{ in } I \textbf{ instances } id\_list \textbf{ do } events_1 \textbf{ end while; skip}]\!]^{D,I,\mathcal{E}}_{while}$$

$$[\![continue]\!]^{D,I,\mathcal{E}}_{event} = [\![continue]\!]^{D,I,\mathcal{E}}_{continue}$$

$$[\![ev]\!]^{D,I,\mathcal{E}}_{events} = [\![ev]\!]^{D,I,\mathcal{E}}_{event} \quad \text{für } ev \text{ keine seq. Komposition}$$

$$[\![ev; events]\!]^{D,I,\mathcal{E}}_{events} = \text{let } \mathcal{E}' = \mathcal{E} \cup bound\_by(ev, I) \text{ in } [\![ev]\!]^{D,I,\mathcal{E}}_{event}; [\![events]\!]^{D,I,\mathcal{E}'}_{events} \quad \text{für } ev \text{ keine Schleife}$$

$$[\![\textbf{loop } ident \textbf{ times instances } id\_list \textbf{ do } events_1 \textbf{ end loop}; events_2]\!]^{D,I,\mathcal{E}}_{events} = [\![\textbf{loop } ident \textbf{ times instances } id\_list \textbf{ do } events_1 \textbf{ end loop}; events_2]\!]^{D,I,\mathcal{E}}_{loop}$$

$$[\![\textbf{while } E \textbf{ in } I \textbf{ instances } id\_list \textbf{ do } events_1 \textbf{ end while}; events_2]\!]^{D,I,\mathcal{E}}_{events} = [\![\textbf{while } E \textbf{ in } I \textbf{ instances } id\_list \textbf{ do } events_1 \textbf{ end while}; events_2]\!]^{D,I,\mathcal{E}}_{while}$$

Abbildung 6.3: Übersetzungsfunktionen für Ereignisse.

$bound\_by(\textbf{in} \ldots \textbf{bind}\ x_1, \ldots, x_n, I)$	=	$\{x_1, \ldots, x_n\}$
$bound\_by(ev, I)$	=	$\emptyset$ für die übrigen $e \in L(comm\_event)$
$bound\_by(\textbf{skip}, I)$	=	$\emptyset$
$bound\_by(\textbf{choose in} \ldots \textbf{bind}\ x_1, \ldots, x_n \textbf{ or } \ldots, I)$	=	$\{x_1, \ldots, x_n\}$
$bound\_by(\textbf{data}\ x_1 = E_1, \ldots, x_n = E_n, I)$	=	$\{x_1, \ldots, x_n\}$
$bound\_by(\textbf{gate}\ m_1 : T_1, \ldots, m_n : T_n, I)$	=	$\{name(chan(m_1, I)), \ldots, name(chan(m_n, I))\}$
$bound\_by(\textbf{select} \ldots, I)$	=	$\emptyset$
$bound\_by(\textbf{loop} \ldots \textbf{do}\ ev\ \textbf{end loop}, I)$	=	$bound\_by(ev)$
$bound\_by(\textbf{while} \ldots \textbf{do}\ ev\ \textbf{end while}, I)$	=	$bound\_by(ev)$
$bound\_by(\textbf{continue} \ldots, I)$	=	$\emptyset$
$bound\_by(ev_1; ev_2, I)$	=	$bound\_by(ev_1, I) \cup bound\_by(ev_2, I)$
$bound\_by(ev, I)$	=	$\emptyset$ für die übrigen $e \in L(event)$

Abbildung 6.4: Funktion zur Berechnung von gebundenen Bezeichnern.

6.4.1 Leeres Ereignis.

Das leere Ereignis **skip** wird durch den erfolgreich terminierten Term **1** dargestellt.

6.4.2 Kommunikationsereignisse.

Zur Angabe der Semantik der Kommunikationsereignisse verwenden wir die Funktion $[\![.]\!]^{D,I,\mathcal{E}}_{comm\_event}$, die in Abbildung 6.5 angegeben ist. Der obere Teil von Abbildung 6.5 enthält die Übersetzung der Sende-Ereignisse, der untere Teil entsprechend die der Empfangsereignisse. Wie bei der Funktion $[\![.]\!]^{\mathcal{E}}_{expr}$ enthält $\mathcal{E}$ die Menge der bisher bekannten Bezeichner der entsprechenden Instanz I.

Bei der Übersetzung der Sendeereignisse unterscheiden wir die einzelnen Nachrichtenarten. Das Senden einer synchronen Nachricht **out** m $(E_1, \ldots, E_n)$ **to** I' wird in eine Ausgabeaktion $m'!E'_1, \ldots, E'_n$ mit $E'_i = [\![E_i]\!]^{\mathcal{E}}_{expr}$ für $1 \leq i \leq n$ und $m' = name(chan(m, I'))$ übersetzt (siehe Definition 6.5). In der Übersetzung ist m' der Bezeichner, an den der das Tor m repräsentierende Kanal $chan(m, I)$ beim Aufruf der aktuellen Prozeßdefinition gebunden wurde. Die Parameter der Nachricht werden durch einen Tupel von Werten realisiert. Handelt es sich um eine asynchrone Nachricht, wird die Ausgabeaktion mit einem *spawn*-Operator umgeben, so daß das Senden der Nachricht parallel zu den nachfolgenden Aktionen der Instanz ausgeführt werden kann. Auf diese Weise wird erreicht, daß der Zeitpunkt der Synchronisation nur von der empfangenden Instanz abhängt und die sendende Instanz unabhängig von der Synchronisation weitere Aktionen

$$[\![\textbf{out } m\_id\ (expr\_list)\ \textbf{to } address]\!]^{D,I,\mathcal{E}}_{comm\_event} = name(chan(m\_id, address))![\![expr\_list]\!]^{\mathcal{E}}_{expr\_list}$$

$$[\![\textbf{out } m\_id\ (expr\_list)\ \textbf{asynch to } address]\!]^{D,I,\mathcal{E}}_{comm\_event} = spawn(name(chan(m\_id, address))![\![expr\_list]\!]^{\mathcal{E}}_{expr\_list})$$

$$[\![\textbf{out } m\_id\ (expr\_list)\ \textbf{call to } address]\!]^{D,I,\mathcal{E}}_{comm\_event} = \textbf{ch } exit_{\$T\$}.\ name(chan(m\_id, address))!([\![expr\_list]\!]^{\mathcal{E}}_{expr\_list}, exit_{\$T\$})$$

$$[\![\textbf{out } m\_id\ (expr\_list)\ \textbf{call asynch to } address]\!]^{D,I,\mathcal{E}}_{comm\_event} = \textbf{ch } exit_{\$T\$}.\ spawn(name(chan(m\_id, address))!([\![expr\_list]\!]^{\mathcal{E}}_{expr\_list}, exit_{\$T\$}))$$

$$[\![\textbf{out } return\ (expr\_list)\ \textbf{return to } address]\!]^{D,I,\mathcal{E}}_{comm\_event} = exit\_address![\![expr\_list]\!]^{\mathcal{E}}_{expr\_list}$$

$$[\![\textbf{out } return\ (expr\_list)\ \textbf{return asynch to } address]\!]^{D,I,\mathcal{E}}_{comm\_event} = spawn(exit_{address}![\![expr\_list]\!]^{\mathcal{E}}_{expr\_list})$$

mit $T\ channel = typeconv(T_2), type\_of(m\_id, I) = T_1\ returns\ T_2$

$$[\![\textbf{in } m\_id\ (expr\_list)\ \textbf{from } address]\!]^{D,I,\mathcal{E}}_{comm\_event} = gen\_input\_nobind(name(chan(m\_id, address)))$$

$$[\![\textbf{in } m\_id\ (expr\_list)\ \textbf{from } address\ \textbf{bind } ident\_list]\!]^{\mathcal{E}}_{comm\_event} = gen\_input\_bind(name(chan(m\_id, address)), ident\_list)$$

$$[\![\textbf{in } m\_id\ (expr\_list)\ \textbf{call from } address]\!]^{D,I,\mathcal{E}}_{comm\_event} = gen\_input\_call(name(chan(m\_id, address)), [], adress)$$

$$[\![\textbf{in } m\_id\ (expr\_list)\ \textbf{call from } address\ \textbf{bind } ident\_list]\!]^{D,I,\mathcal{E}}_{comm\_event} = gen\_input\_call(name(chan(m\_id, address)), ident\_list, address)$$

$$[\![\textbf{in } m\_id\ (expr\_list)\ \textbf{return from } address]\!]^{D,I,\mathcal{E}}_{comm\_event} = gen\_input\_nobind(exit)$$

$$[\![\textbf{in } m\_id\ (expr\_list)\ \textbf{return from } address\ \textbf{bind } ident\_list]\!]^{D,I,\mathcal{E}}_{comm\_event} = gen\_input\_bind(exit, ident\_list)$$

Abbildung 6.5: Übersetzungsfunktionen für Kommunikationsereignisse.

$gen\_input\_nobind\,(m\_id) =$
$\quad m\_id?\_value; \mathbf{1}$

$gen\_input\_bind\,(m\_id, ident\_list) =$
$\quad$ if $ident\_list = x$ then
$\quad\quad m\_id?x; \mathbf{1}$
$\quad$ else
$\quad\quad m\_id?\_value; \{x_1 \leftarrow \#1\ \_value, ..., x_n \leftarrow \#n\ \_value\}; \mathbf{1}$

$gen\_input\_call\,(m\_id, ident\_list, address) =$
$\quad m\_id?\_value; \{x_1 \leftarrow \#1\ \_value, ..., x_n \leftarrow \#n\ \_value, exit_{address} \leftarrow \#(n+1)\ \_value\}; \mathbf{1}$

Abbildung 6.6: Hilfsfunktionen zur Zerlegung von Tupeln.

durchführen kann.

Das Senden einer Anfrage **out** m $(E_1, ..., E_n)$ **call to** I' wird in eine Aktion **ch** $exit_{\$T\$}.\, m'!(E'_1, ..., E'_n, exit_{\$T\$})$ übersetzt. Es wird ein neuer Kanal $exit_{\$T\$}$ für den Erhalt des Rückgabewerts der Anfrage erzeugt, wobei der Typ $T\ channel = typeconv(T_2)$ für $type\_of(m, I) = T_1\ returns\ T_2$ ist. Die Aktion überträgt auf dem Kanal neben den Parametern auch den *exit*-Kanal. Bei einer asynchronen Anfrage wird das Senden auf dem Kanal wieder durch einen *spawn*-Operator in einem eigenen Prozeß realisiert.

Das Senden einer Rückantwort **out** m $(E_1, ..., E_n)$ **return to** I verwendet den zuvor von der anfragenden Instanz I' erzeugten Rückgabekanal. Dieser wird von der die Anfrage bearbeitenden Instanz unter einem Bezeichner $exit_{I'}$ gespeichert (s.u.), so daß das Senden in eine Aktion $exit_{I'}!E'_1, ..., E'_n$ übersetzt wird. Auch in diesem Fall wird eine asynchrone Nachricht durch das Einbetten in einen *spawn*-Operator realisiert.

Die Empfangsereignisse der Diagramme werden in Eingabeaktionen von $\mathcal{P}$ übersetzt. Dabei verwenden wir die in Abbildung 6.6 angegeben Funktionen zur Erzeugung der unterschiedlichen Eingabeaktionen. Hierbei unterscheiden wir zum einen, ob Bezeichner angegeben sind, an die die empfangenen Werte gebunden werden sollen. Zum anderen unterscheiden wir zwischen Anfragen und Nachrichten ohne Rückantwort.

Die Semantik des Empfangs einer Nachricht **in** $m(E_1, ..., E_n)$ **from** I' ohne die Bindung der empfangenen Werte an Bezeichner ist der Prozeß $m'?\_value; \mathbf{1}$ mit $m' = name(chan(m, I'))$. Über den an den Bezeichner m' gebundenen Kanal wird ein Tupel von Werten empfangen und an den Bezeichner *_value* gebunden, der hierbei nur als "Dummy"-Variable fungiert. Der zusätzliche **1**-Operator wird vorsorglich eingefügt, falls die Eingabeaktion die letzte Aktion einer Instanz ist.

Da die Eingabe ein Präfixoperator ist, muß auch in diesem Fall ein Term angegeben werden, der den Gültigkeitsbereich des Bezeichners *_value* darstellt. Folgen auf den Empfang der Nachricht weitere Aktionen, kann der nun überflüssige **1**-Prozeß durch Anwendung der Axiome (3.3) und (3.4) aus Abbildung 3.4 aus dem Term entfernt werden. In Kapitel 3 haben wir festgelegt, daß sequentielle Komposition eine höhere Priorität als Restriktion besitzt, daher entspricht $E?x; t_1; t_2$ dem Term $E?x; (t_1; t_2)$. Unsere Semantik erzeugt nun für Eingabeereignisse einen Term $E?x; \mathbf{1}; t_0$, wobei t_0 der Term für die nachfolgenden Ereignisse auf der Lebenslinie der Instanz ist (siehe $[\![.]\!]^{D,I,\mathcal{E}}_{events}$ in Abbildung 6.3). Ist t_0 nicht leer, kann also der **1**-Operator durch Anwendung von (3.3) aus dem Term entfernt werden: $E?x; (\mathbf{1}; t_0) = E?x; t_0$.

Handelt es sich um ein Empfangsereignis **in** $m(E_1, ..., E_n)$ **from** I' **bind** $x_1, ..., x_n$, unterscheiden wir in *gen_input_bind* zwei Fälle. Wird nur ein Parameter übertragen, übersetzen wir die Aktion in den Term $m'?x; \mathbf{1}$, in dem die Bindung direkt durch die Eingabeaktion durchgeführt wird. Werden mehr als ein Parameter übertragen, erzeugen wir einen Term

$$m'?\_value; \{x_1 \leftarrow \#1\ \_value, ..., x_n \leftarrow \#n\ \_value\}; \mathbf{1},$$

in dem der übertragene Tupel von Parametern an *_value* gebunden und anschließend durch einen Definitionsoperator in seine Komponenten zerlegt wird. Diese Komponenten werden dabei an die angegebenen Bezeichner gebunden. Auch hier wird ein abschließender **1**-Term angegeben, da auch der Definitionsoperator ein Präfixoperator ist.

Handelt es sich bei der Eingabe um den Empfang einer Anfrage, wird zusätzlich die $(n+1)$-te Komponente des Tupels, die den Rückgabekanal für die Antwort enthält, an den impliziten Bezeichner $exit_{address}$ gebunden, wobei *address* der Name der Instanz ist, von der die Anfrage gesendet wurde. Der Term hierfür wird von der Funktion *gen_input_bind* generiert.

Beim Empfang der Rückantwort wird wieder der Kanal *exit* verwendet, der beim Senden der Anfrage dynamisch erzeugt wurde. Somit wird eine vollständige Anfrage auf der Senderseite der Form

out $call(E_1, ..., E_n)$ **call to** J; **in** $return(E'_1, ..., E'_m)$ **return from** J **bind** $x_1, ..., x_n$

in den folgenden Prozeß übersetzt:

$$\begin{aligned} &\textbf{ch}\ exit_{\$T\$}.\ call!(F_1, ..., F_n, exit_{\$T\$}); \\ &\qquad exit_{\$T\$}?\_value; \{x_1 \leftarrow \#1\ \_value, ..., x_m \leftarrow \#m\ \_value\}; \mathbf{1}, \end{aligned}$$

wobei F_i die Übersetzung des jeweiligen Ausdrucks E_i, $1 \leq i \leq n$ ist.

6.4.3 Auswahl zwischen mehreren Empfangsaktionen

Die Auswahl zwischen mehreren Empfangsaktionen mit **choose** wird in Abbildung 6.3 übersetzt, indem wir das syntaktisch erste Ereignis an die Funktion

$[\![.]\!]^{D,I,\mathcal{E}}_{comm\_event}$ weitergeben. Alle Eingabeereignisse nach einem **choose** verwenden dasselbe Tor und binden gegebenenfalls dieselben Bezeichner. Da nur eines der möglichen Eingabeereignisse eintreten kann, ist es ausreichend, alle möglichen Eingabeereignisse durch eine einzige Eingabeaktion aus $\mathcal{P}$ darzustellen. Wir übersetzen daher nur das erste Ereignis. Eine Auswahl der Form

choose
  in $m(v_1, v_2)$ **from** I_1 **bind** x_1, x_2
or
  in $m(w_1, w_2)$ **from** I_2 **bind** x_1, x_2

wird somit in einen Prozeß

$$m'?\_value; \{x_1 \leftarrow \#1\ \_value, x_2 \leftarrow \#2\ \_value\}; \mathbf{1}$$

übersetzt. Diese semantische Modellierung ist insbesondere möglich, da die Adressierung der Instanzen allein über die Kommunikationskanäle geschieht und somit die sendende Instanz in der erzeugten Übersetzung nicht angegeben wird.

6.4.4 Definition von Daten.

Die Semantik der Definition von Daten geben wir mit der Funktion $[\![.]\!]^{D,I,\mathcal{E}}_{data}$ in Abbildung 6.7 an. Die definierten Bezeichner und die Ausdrücke zur Berechnung der an sie zu bindenden Werte werden mit Hilfe des Definitionsoperators von $\mathcal{P}$ realisiert. Während in einer Definition **data** $x_1 = E_1, ..., x_n = E_n$ ein Bezeichner x_i in E_j mit $j > i$ auftreten darf, ist dies im Definitionsoperator $\{x_1 \leftarrow E_1, ..., x_n \leftarrow E_n\}; t$ nicht möglich (siehe Seite 125). Daher muß jede Definition $x_i = E_i$ durch einen eigenen Definitionsoperator realisiert werden.

Da der Definitionsoperator ein Präfixoperator ist, geben wir wie bei der Übersetzung der Eingabeereignisse einen abschließenden **1**-Operator an, um eine Fallunterscheidung bzgl. nachfolgender Ereignisse zu vermeiden (s.o.).

6.4.5 Erzeugung von Kanälen.

Die **gate**-Anweisung zur Generierung neuer Kanäle wird mit der in Abbildung 6.7 angegebenen Funktion $[\![.]\!]^{D,I,\mathcal{E}}_{gate}$ übersetzt. Für jedes neue Tor m_i, das Nachrichten des Typs T_i empfangen kann, erzeugen wir dynamisch mit dem **ch**-Operator einen neuen Kanal $c_{i\$T'_i\$}$ des Typs $typeconv(T_i) = T'_i\ channel$. Diese Kanäle werden dann in einem Deklarationsoperator an Bezeichner $c_i = name(c_{i\$T'_i\$})$ gebunden, so daß auf die Kanäle für die initialen Tore und die Kanäle für dynamisch erzeugte Tore einheitlich zugegriffen werden kann. Der abschließende **1**-Operator

$$[\![\textbf{data}\, x_1 = E_1, x_2 = E_2, ..., x_n = E_n]\!]^{D,I,\mathcal{E}}_{data} =$$
$$\{x_1 \leftarrow [\![E_1]\!]^{\mathcal{E}}_{expr}\};$$
$$\{x_2 \leftarrow [\![E_2]\!]^{\mathcal{E}}_{expr}\};$$
$$...$$
$$\{x_n \leftarrow [\![E_n]\!]^{\mathcal{E}}_{expr}\}; \mathbf{1}$$

$$[\![\textbf{gate}\, m_1 : T_1, ..., m_n : T_n]\!]^{D,I,\mathcal{E}}_{gate} =$$
$$\textbf{ch}\ c_{1\$T'_1\$}, ..., c_{n\$T'_n\$}.\ \{c_1 \leftarrow c_{1\$T'_1\$}, ..., c_n \leftarrow c_{n\$T'_n\$}\}; \mathbf{1}$$

mit $chan(m_1, I) = c_{1\$T'_1\$}, ..., chan(m_n, I) = c_{n\$T'_n\$}$
$typeconv(T_i) = T'_i\ channel$

Abbildung 6.7: Übersetzungsfunktion für Datendeklarationen und Kanalerzeugung.

wird wie bei der Übersetzung der Eingabeereignisse zur Vermeidung der Fallunterscheidung bzgl. nachfolgender Aktionen verwendet (s.o.).

Als Beispiel für die Übersetzung einer Kanalerzeugung geben wir die Semantik der Annotationen aus Abbildung 5.10 auf Seite 128 unter den Annahmen $chan(m, I_1) = m_{1\$int\$}$ und $chan(m_r, I_1) = m_{r\$int \times bool\ channel\$}$ an; sie lauten

$$\textbf{ch}\ m_{\$int\$}.\ \{m \leftarrow m_{\$int\$}\}; \mathbf{1}$$

und

$$\textbf{ch}\ m_{r\$int \times bool\ channel\$}.\ \{m_r \leftarrow m_{r\$int \times bool\ channel\$}\}; \mathbf{1}.$$

6.4.6 Alternativen.

Für die Übersetzung der Auswahl zwischen mehreren Alternativen verwenden wir die in Abbildung 6.8 angegebenen Funktionen. Die Funktion $[\![.]\!]^{D,I,\mathcal{E}}_{choice}$ übersetzt die einzelnen Alternativen mit der $[\![.]\!]^{D,I,\mathcal{E}}_{alt}$-Funktion und verknüpft die übersetzten Terme mit Auswahloperatoren. Der erzeugte Term wird mit Klammern umgeben, um die korrekte Schachtelung der Teilterme im Gesamtterm zu gewährleisten, da sequentielle Komposition eine höhere Priorität als der Auswahloperator hat.

Alternativen werden von der Funktion $[\![.]\!]^{D,I,\mathcal{E}}_{alt}$ in Terme mit einem führenden bedingten Operator [.] übersetzt, der die Gültigkeit der Bedingung überprüft.

$$[\![\textbf{select } alt_1 \ldots alt_n]\!]^{D,I,\mathcal{E}}_{choice} = ([\![alt_1]\!]^{D,I,\mathcal{E}}_{alt} + \ldots + [\![alt_n]\!]^{D,I,\mathcal{E}}_{alt})$$

$$[\![\textbf{if } expr\ events]\!]^{D,I,\mathcal{E}}_{alt} = [[\![expr]\!]^{\mathcal{E}}_{expr}]; [\![events]\!]^{D,I,\mathcal{E}}_{events}$$

Abbildung 6.8: Übersetzungsfunktionen für Alternativen.

Die Ereignisse der entsprechenden Alternative werden dann mit der Funktion $[\![.]\!]^{D,I,\mathcal{E}}_{events}$ in einen Prozeßterm übersetzt. Somit wird eine Alternative **select if** E_1 $events_1$ ... **if** E_n $events_n$ in einen Term $([E'_1]; t_1 + \ldots + [E'_n]; t_n)$ übersetzt, wobei der Term t_i die Übersetzung der Ereignisfolge $events_i$ der i-ten Alternative darstellt $(1 \leq i \leq n)$.

6.4.7 Schleifen.

Zur Übersetzung von Schleifen verwenden wir die Übersetzungsfunktionen $[\![.]\!]^{D,I,\mathcal{E}}_{loop}$ für *loop*-Schleifen und $[\![.]\!]^{D,I,\mathcal{E}}_{while}$ für *while*-Schleifen. Beide Funktionen fügen zusätzliche Prozeßdefinitionen in die Prozeßumgebung Θ ein, da zur Realisierung der Schleifen die Schleifeniteration durch rekursive Prozesse dargestellt wird. Dabei wird für jede Schleife, an der eine Instanz beteiligt ist, eine neue Prozeßdefinition für diese Instanz erzeugt.

loop-Schleifen. Die Funktion $[\![.]\!]^{D,I,\mathcal{E}}_{loop}$ für Schleifen ist in Abbildung 6.9 angegeben. Eine Schleife ist ein imperatives Konstrukt, da in jedem Iterationsschritt neue Werte an die im Schleifenrumpf deklarierten Bezeichner gebunden werden. Da der Kalkül $\mathcal{P}$ keine imperativen Konstrukte direkt unterstützt, bilden wir die Iteration des Schleifeninhalts durch Endrekursion nach. Hierzu ist es nötig, für jede Schleife eine neue Prozeßdefinition in die Prozeßumgebung Θ einzufügen, die die Semantik des Schleifeninhalts realisiert. Die Prozeßdefinition besitzt die folgende Struktur:

$$D\_I\_loop\_l(\_count : int, x_1 : T_1, \ldots, x_k : T_k) \mapsto$$
$$[\_count > 0]; t_1; D\_I\_loop\_l(\_count - 1, E_1, \ldots, E_k)$$
$$+[\_count \leq 0]; t_2$$

Der Name *D_I_loop_l* des Prozesses enthält den Namen des D des Diagramms, der jeweiligen Instanz I sowie eine laufende Nummer l zur Unterscheidung der einzelnen Prozeßdefinitionen, falls eine Instanz in einem Diagramm an mehr als einer Schleife beteiligt ist. Wir nehmen an, daß eine Funktion $gen\_number(I)$ existiert, die bei jedem Aufruf eine neue Nummer für die Instanz I generiert.

Als ersten Parameter des Prozesses verwenden wir eine spezielle Zählvariable *_count*, die die Anzahl der noch durchzuführenden Iterationen enthält. Der Prozeß führt anfänglich eine Fallunterscheidung durch. Ist der Wert von *_count* größer als 0, wird eine weitere Iteration durchgeführt, andernfalls wird die Ausführung nach der Schleife fortgesetzt. In der ersten Alternative repräsentiert t_1 die Übersetzung des Schleifeninhalts. Nach der Ausführung von t ruft sich der Prozeß rekursiv auf. Hierbei wird der Wert von *_count* dekrementiert. Die Ausdrücke E_1 bis E_k enthalten neue Argumente für die Parameter $x_1, ..., x_k$. In der zweiten Alternative ist t_2 die Übersetzung der auf die Schleife folgenden Sequenz von Ereignissen der Instanz.

Die Parameterliste des Prozesses enthält neben dem Bezeichner *_count* einen Vektor *typedvect* der Form $x_{f1} : T_{f1}, ..., x_{fk} : T_{fk}$, der die für die Ausführung von t_1 und t_2 notwendigen Bezeichner x_i enthält. Dieser Vektor ist lexikographisch geordnet und enthält die folgenden beiden Arten von Bezeichnern:

- Die erste Art von Bezeichnern sind die Bezeichner, die vor dem Eintreten in die Schleife definiert wurden. Diese Bezeichner entnimmt die Übersetzungsfunktion der Menge $\mathcal{E}$, die die bisher bekannten Bezeichner enthält. Beim rekursiven Aufruf von *D_I_loop_l* werden die an diese Bezeichner gebundenen Werte als Argumente übergeben.

- Die zweite Art sind die während der Schleife definierten Bezeichner. Diese werden mit der Funktion *bound_by* ermittelt. Da diese Bezeichner auch beim initialen Aufruf von *D_I_loop_l* Werte erhalten müssen, wird bei diesem Aufruf für jeden dieser Bezeichner als "Dummy" ein beliebiger Wert aus dem Wertebereich des Bezeichners als Argument übergeben. Zu dieser Art von Bezeichnern gehören auch die innerhalb der Schleife erzeugten lokalen Tore, da die zugehörigen Bezeichner in der Funktion *bound_by* in Abbildung 6.4 berücksichtigt werden.

Da der Term $t_2 = [\![ev_2]\!]_{events}^{D,I,\mathcal{E} \cup bound\_by(ev_1,I)}$ sich im Gültigkeitsbereich der Parameter $x_1, ..., x_k$ befindet, ist gewährleistet, daß die in der Schleife erzeugten Bezeichner auch nach der Schleife in t_2 sichtbar sind.

Soll in Diagrammen das Verschachteln von *loop*-Schleifen möglich sein, muß anstelle der einzelnen Variable *_count* ein Vektor oder eine Liste von Zählvariablen verwendet werden, bei der der Index eines Vektor bzw. Listenelements die entsprechende Schachtelungstiefe darstellt (siehe Abschnitt 6.7).

$[\![$**loop** *ident* **times instances** $J_1, ..., J_l$ **do** ev_1 **end loop**; $ev_2]\!]^{D,I,\mathcal{E}}_{loop} =$
  let $\_ = include\_\Theta(definition)$ in
    $D\_I\_loop\_l([\![ident]\!]^{\mathcal{E}}_{expr}, params)$

mit
$definition =$
  $D\_I\_loop\_l(\_count : int, typedvect) \mapsto$
    $[\_count > 0];$
      $[\![ev_1]\!]^{D,I,\mathcal{E}}_{events};$
      $D\_I\_loop\_l(\_count - 1, vect)$
    $+[\_count \leq 0];$
      $[\![ev_2]\!]^{D,I,\mathcal{E} \cup bound\_by(ev_1,I)}_{events}$

$l = gen\_number(I)$,
$vars = \mathcal{E} \cup bound\_by(ev_1, I) = \{x_1, ..., x_k\}$,
$vect = make\_vect(\{x_1, ..., x_k\}) = \{x_{f1}, ..., x_{fk}\}$
$typedvect = x_{f1} : T_{x1}, ..., x_{fk} : T_{fk}$
  mit $T_{fi} = typeconv(type\_of(x_{fi}))$
$params = E_1, ..., E_k$
  mit $E_i = x_{fi}$ für $x_{fi} \in \mathcal{E}$
  und $E_i = v, v$ beliebig aus $\textsc{Value}^{type\_of(x_{fi})}$,
    für $x_{fi} \in bound\_by(ev_1, I)$

Abbildung 6.9: Übersetzungsfunktion für *loop*–Schleifen

$[\![$**while** E **in** I' **instances** $J_1, ..., J_l$ **do** ev_1 **end while**; $ev_2]\!]^{D,I,\mathcal{E}}_{while} =$
let $\_ = include\_\Theta(definition)$ in

$D\_I\_while\_l(params)$

mit
$definition =$
if $I \neq I'$ then
$D\_I\_while\_l(typedvect) \mapsto$
$\_while_I?\_continue;$
$([\_continue];$

$[\![ev_1]\!]^{D,I,\mathcal{E}}_{events};$

$D\_I\_while\_l(vect)$

$+[not\ \_continue];$

$[\![ev_2]\!]^{D,I,\mathcal{E} \cup bound\_by(ev_1,I)}_{events})$

else
$D\_I\_while\_l(typedvect) \mapsto$
$\{\_continue \leftarrow [\![E]\!]^{\mathcal{E}}_{expr}\};$
$\_while_{J_{i_1}}!\_continue;$
$...;$
$\_while_{J_{i_k}}!\_continue;$
$([\_continue];$

$[\![ev_1]\!]^{D,I,\mathcal{E}}_{events};$

$D\_I\_while\_l(vect)$

$+[not\ \_continue];$

$[\![ev_2]\!]^{D,I,\mathcal{E} \cup bound\_by(ev_1,I)}_{events})$

$\{J_{i_1}, ..., J_{i_k}\} = \{J_1, ..., J_l\} \setminus \{I\}$,
$l, vars, vect, typedvect$ und $params$ wie in Abbildung 6.9

Abbildung 6.10: Übersetzungsfunktion für *while*–Schleifen

Für die Angabe eines Beispiels zur Semantik von *loop*-Schleifen verweisen wir auf die Semantik der Gefangenendilemma-Spezifikation in Abschnitt 6.5.

while-Schleifen. Die Funktion $[\![.]\!]^{D,I,\mathcal{E}}_{while}$ zur Übersetzung von *while*-Schleifen ist in Abbildung 6.10 angegeben. Analog zu den Prozeßdefinitionen $D\_I\_loop\_l$ bei *loop*-Schleifen fügen wir nun Prozeßdefinitionen $D\_I\_while\_l$ in die Prozeßumgebung Θ ein. Während bei *loop*-Schleifen jede Instanz selbst über die Fortführung der Schleife entscheidet, kontrolliert bei einer *while*-Schleife eine spezielle Instanz die Schleifenbedingung. Daher unterscheiden wir in der Funktion in Abbildung 6.10 zwei Fälle: Im ersten Fall wird eine Instanz I übersetzt, die nicht die Kontrollinstanz I' ist. Die Prozeßdefinition $D\_I\_while\_l$ empfängt einen booleschen Wert über den Kanal $\_while_I$ zur Schleifensteuerung und bindet ihn an den Bezeichner *_continue*. Wurde der Wert *true* gesendet, wird eine weitere Iteration der Schleife durchgeführt und die Prozeßdefinition rekursiv aufgerufen. Andernfalls terminiert die Schleife und die auf die Schleife folgenden Aktionen werden ausgeführt.

Handelt es sich bei der Instanz um die Kontrollinstanz I', wird zu Beginn von $D\_I\_while\_l$ der Ausdruck für die Schleifenbedingung ausgewertet und an den Bezeichner *_continue* gebunden. Anschließend wird der Wert von *_continue* an alle anderen an der Schleife beteiligten Instanzen über den jeweiligen $\_while_I$-Kanal gesendet. Danach wird in einer Fallunterscheidung anhand des Wertes von *_continue* entschieden, ob eine weitere Iteration durchgeführt werden soll. Ist dies der Fall, wird der Schleifenrumpf ausgeführt und anschließend die Prozeßdefinition $D\_I\_while\_l$ aufgerufen. Andernfalls werden die Aktionen nach dem Schleifenende ausgeführt.

Die Funktion $[\![.]\!]^{D,I,\mathcal{E}}_{while}$ verwendet die in Abbildung 6.9 angegebenen Hilfsdefinitionen. Die $\_while_I$-Kanäle treten in den Parameterlisten der Prozeßdefinitionen nicht explizit auf, da die Kanäle zu Beginn der Ausführung des Systems erzeugt wurden und die zugehörigen Bezeichner bereits in der Namensumgebung $\mathcal{E}$ enthalten sind. Daher sind sie auch Bestandteil von *typedvect* und *params*.

6.5 Semantik des Gefangenendilemma-Beispiels

In diesem Abschnitt geben wir die Semantik des Gefangenendilemma-Beispiels aus Abschnitt 5.3.5 an. Hierzu übersetzen wir die textuelle Darstellung des Diagramms aus Beispiel 5.6 auf Seite 152 in den Kalkül $\mathcal{P}_\lambda$. Im folgenden ist die Übersetzung des Beispiels angegeben:

Prozeßumgebung Θ:

$Gefangenendilemma\_Spieler\_A\_start($
$\quad ergebnis : (int \times int)\ channel, gegner_A : bool\ channel, gegner_B : bool\ channel,$

$runden_A : int\ channel, runden_B : int\ channel, runden_S : int\ channel,$
$wahl_A : bool\ channel, wahl_B : bool\ channel$
$) \mapsto$
$runden_A?n;$
$wahl_A!true;$
$Gefangenendilemma\_Spieler\_A\_loop\_1$
$(n, ergebnis, gegner_A, gegner_B, n, runden_A, runden_B, runden_S,$
$wahl_A, wahl_B, false)$

$Gefangenendilemma\_Spieler\_A\_loop\_1($
$\_count : int, ergebnis : (int \times int)\ channel,$
$gegner_A : bool\ channel, gegner_B : bool\ channel, n : int,$
$runden_A : int\ channel, runden_B : int\ channel, runden_S : int\ channel,$
$wahl_A : bool\ channel, wahl_B : bool\ channel, zuletzt : bool,$
$) \mapsto$
$[\_count > 0];$
$gegner?zuletzt;$
$wahl\_A!zuletzt;$
$Gefangenendilemma\_Spieler\_A\_loop\_1($
$\_count - 1, ergebnis, gegner_A, gegner_B, n, runden_A, runden_B, runden_S,$
$wahl_A, wahl_B, zuletzt)$
$+[count \leq 0]$

$Gefangenendilemma\_Schiedsrichter\_start($
$ergebnis : (int \times int)\ channel, gegner_A : bool\ channel, gegner_B : bool\ channel,$
$runden_A : int\ channel, runden_B : int\ channel, runden_S : int\ channel,$
$wahl_A : bool\ channel, wahl_B : bool\ channel$
$) \mapsto$
$\{auswerten \leftarrow$
$\lambda x : bool.\lambda y : bool.$
$if\ x\ and\ y\ then\ 3$
$else\ if\ x\ and\ not\ y\ then\ 5$
$else\ if\ not\ x\ and\ y\ then\ 0$
$else\ if\ not\ x\ and\ not\ y\ then\ 1\};$
$\{summe_A \leftarrow 0\};$
$\{summe_B \leftarrow 0\};$
$runden_S?n;$
$runden_A!n;$
$runden_B!n;$
$wahl_A?w_A;$
$wahl_B?w_B;$
$Gefangenendilemma\_Schiedsrichter\_loop\_1$
$(n, auswerten, ergebnis, gegner_A, gegner_B, runden_A, runden_B, runden_S,$

$$0, 0, \mathit{false}, \mathit{false}, \mathit{wahl}_A, \mathit{wahl}_B)$$

$$
\begin{array}{l}
\mathit{Gefangenendilemma\_Schiedsrichter\_loop\_1}(\\
\quad \mathit{\_count} : \mathit{int}, \mathit{auswerten} : \mathit{bool} \to \mathit{bool} \to \mathit{int}, \mathit{ergebnis} : (\mathit{int} \times \mathit{int})\ \mathit{channel}, \\
\quad \mathit{gegner}_A : \mathit{bool}\ \mathit{channel}, \mathit{gegner}_B : \mathit{bool}\ \mathit{channel}, \mathit{runden}_A : \mathit{int}\ \mathit{channel}, \\
\quad \mathit{runden}_B : \mathit{int}\ \mathit{channel}, \mathit{runden}_S : \mathit{int}\ \mathit{channel}, \mathit{summe}_A : \mathit{int}, \mathit{summe}_B : \mathit{int}, \\
\quad w_A : \mathit{bool}, w_B : \mathit{bool}, \mathit{wahl}_A : \mathit{bool}\ \mathit{channel}, \mathit{wahl}_B : \mathit{bool}\ \mathit{channel} \\
) \ \mapsto \\
\quad [\mathit{\_count} > 0]; \\
\qquad \{\mathit{summe}_A \leftarrow \mathit{summe}_A + \mathit{auswerten}\ w_A\ w_B\}; \\
\qquad \{\mathit{summe}_B \leftarrow \mathit{summe}_B + \mathit{auswerten}\ w_B\ w_A\}; \\
\qquad \mathit{gegner}_A!w_B; \\
\qquad \mathit{gegner}_B!w_A; \\
\qquad \mathit{wahl}_A?w_A; \\
\qquad \mathit{wahl}_B?w_B; \\
\qquad \mathit{Gefangenendilemma\_Schiedsrichter\_loop\_1} \\
\qquad\quad (\mathit{\_count} - 1, \mathit{auswerten}, \mathit{ergebnis}, \mathit{gegner}_A, \mathit{gegner}_B, \mathit{runden}_A, \mathit{runden}_B, \\
\qquad\qquad \mathit{runden}_S, \mathit{summe}_A, \mathit{summe}_B, w_A, w_B, \mathit{wahl}_A, \mathit{wahl}_B) \\
\quad +[\mathit{\_count} \leq 0]; \\
\qquad \{\mathit{summe}_A \leftarrow \mathit{summe}_A + \mathit{auswerten}\ w_A\ w_B\}; \\
\qquad \{\mathit{summe}_B \leftarrow \mathit{summe}_B + \mathit{auswerten}\ w_B\ w_A\}; \\
\qquad \mathit{ergebnis}!\mathit{summe}_A, \mathit{summe}_B
\end{array}
$$

$$
\begin{array}{l}
\mathit{Gefangenendilemma\_Spieler\_B\_start}(\\
\quad \mathit{ergebnis} : (\mathit{int} \times \mathit{int})\ \mathit{channel}, \mathit{gegner}_A : \mathit{bool}\ \mathit{channel}, \mathit{gegner}_B : \mathit{bool}\ \mathit{channel}, \\
\quad \mathit{runden}_A : \mathit{int}\ \mathit{channel}, \mathit{runden}_B : \mathit{int}\ \mathit{channel}, \mathit{runden}_S : \mathit{int}\ \mathit{channel}, \\
\quad \mathit{wahl}_A : \mathit{bool}\ \mathit{channel}, \mathit{wahl}_B : \mathit{bool}\ \mathit{channel} \\
) \ \mapsto \\
\quad \{\mathit{zuege} \leftarrow []\}; \\
\quad \{\mathit{anzahl} \leftarrow \\
\qquad \mathit{let}\ \mathit{anzahl}_0 = \\
\qquad\quad \lambda f : (\mathit{bool}\ \mathit{list} \to \mathit{int}) \to \mathit{bool}\ \mathit{list} \to \mathit{int}. \\
\qquad\quad \lambda l : \mathit{bool}\ \mathit{list}. \\
\qquad\qquad \mathit{if}\ l = []\ \mathit{then}\ 0\ \mathit{else}\ 1 + f(\mathit{tail}\ l) \\
\qquad \mathit{in}\ Y\ \mathit{anzahl}_0\}; \\
\quad \{\mathit{betrogen} \leftarrow \\
\qquad \mathit{let}\ \mathit{betrogen}_0 = \\
\qquad\quad \lambda f : (\mathit{bool}\ \mathit{list} \to \mathit{int}) \to \mathit{bool}\ \mathit{list} \to \mathit{int}. \\
\qquad\quad \lambda l : \mathit{bool}\ \mathit{list}. \\
\qquad\qquad \mathit{if}\ l = []\ \mathit{then}\ 0 \\
\qquad\qquad \mathit{else\ if}\ \mathit{head}\ l\ \mathit{then}\ f(\mathit{tail}\ l)\ \mathit{else}\ 1 + f(\mathit{tail}\ l) \\
\qquad \mathit{in}\ Y\ \mathit{betrogen}_0\}; \\
\quad \mathit{runden}_B?n;
\end{array}
$$

$wahl_B!false;$
$Gefangenendilemma\_Spieler\_B\_loop\_1$
$(n, anzahl, betrogen, ergebnis, gegner_A, gegner_B, n, false, runden_A,$
$runden_B, runden_S, wahl_A, wahl_B, zuege, false)$

$Gefangenendilemma\_Spieler\_B\_loop\_1($
$\_count : int, anzahl : bool\ list \to int, betrogen : bool\ list \to int,$
$ergebnis : (int \times int)\ channel, gegner_A : bool\ channel, gegner_B : bool\ channel,$
$n : int, neu : bool, runden_A : int\ channel, runden_B : int\ channel,$
$runden_S : int\ channel, wahl_A : bool\ channel, wahl_B : bool\ channel,$
$zuege : bool\ list, zuletzt : bool$
$) \mapsto$
$[\_count > 0];$
$gegner_B?zuletzt;$
$\{zuege \leftarrow cons\ zuletzt\ zuege\};$
$\{neu \leftarrow$
$let\ b = betrogen\ zuege$
$in\ not(anzahl\ zuege - b < b)\};$
$wahl_B!neu;$
$Gefangenendilemma\_Spieler\_B\_loop\_1$
$(\_count - 1, anzahl, betrogen, ergebnis, gegner_A, gegner_B, neu, runden_A,$
$runden_B, runden_S, n, zuletzt, wahl_A, wahl_B, zuege)$
$+[\_count \leq 0]$

Initialer Prozeßterm:

$\mathbf{ch}\ \{gegner_{A\$bool\$}, gegner_{B\$bool\$}, runden_{A\$int\$}, runden_{B\$int\$},$
$wahl_{A\$bool\$}, wahl_{B\$bool\$}\}.$
$spawn(Gefangenendilemma\_Spieler\_A\_start($
$ergebnis_{\$int\times int\$}, gegner_{A\$bool\$}, gegner_{B\$bool\$}, runden_{A\$int\$},$
$runden_{B\$int\$}, runden_{S\$int\$}, wahl_{A\$bool\$}, wahl_{B\$bool\$}));$
$spawn(Gefangenendilemma\_Schiedsrichter\_start($
$ergebnis_{\$int\times int\$}, gegner_{A\$bool\$}, gegner_{B\$bool\$}, runden_{A\$int\$},$
$runden_{B\$int\$}, runden_{S\$int\$}, wahl_{A\$bool\$}, wahl_{B\$bool\$}));$
$spawn(Gefangenendilemma\_Spieler\_B\_start($
$ergebnis_{\$int\times int\$}, gegner_{A\$bool\$}, gegner_{B\$bool\$}, runden_{A\$int\$},$
$runden_{B\$int\$}, runden_{S\$int\$}, wahl_{A\$bool\$}, wahl_{B\$bool\$}));$

Die Prozeßumgebung Θ enthält sechs Prozeßdefinitionen: Für jede Instanz I der drei Instanzen gibt es einen Prozeßdefinition I_start, mit der die Ausführung der Instanz beginnt. Die drei Prozeßdefinitionen mit der Endung $\_loop\_1$ realisieren die Ausführung der Schleife in den einzelnen Instanzen. Wie bereits in Beispiel

6.9 auf Seite 175 angegeben, erhalten die *_start*-Prozesse als Parameter die global bekannten Kanalwerte. Die Reihenfolge der Parameter wird durch die lexikographische Ordnung der Kanalbezeichner festgelegt. Die *_loop_1*-Prozesse erwarten als Parameter die globalen Kanäle sowie alle Bezeichner, die vor der Schleife deklariert wurden. Ebenso werden die in der Schleife deklarierten Bezeichner in der Parameterliste angegeben, da auf diese Bezeichner in den einzelnen Iterationen und nach Beendigung der Schleife zugegriffen werden kann. Beim ersten Aufruf der *_loop_1*-Prozesse aus den *_start*-Prozessen werden diese Parameter mit beliebigen Werten aus dem jeweiligen Wertebereich des Bezeichners belegt: Für Parameter des Typs *bool* übergeben wir *false*, für Parameter des Typs *int* übergeben wir 0.

Der initiale Prozeßterm ist aus Beispiel 6.9 übernommen. Allerdings haben wir in diesem Term und in den Parameterlisten der Prozeßdefinitionen die $\_cont_I$- und $\_while_I$-Kanäle entfernt, da diese im Beispiel nicht benötigt werden (siehe Axiom (3.16) in Abbildung 3.4). Diese Optimierung verbessert die Lesbarkeit der Prozesse. Die Kanäle für Interaktion zwischen den Instanzen werden durch einen Restriktionsoperator dynamisch erzeugt. Außerdem wird durch die Restriktion die Kommunikation über die Kanäle erzwungen (siehe R_{15} und R_{16} in Abbildung 3.3). Die Kanäle für die Interaktion mit der Umgebung werden nicht restringiert, aber wie die übrigen Kanäle in den Prozeßdefinitionen als Parameter übergeben.

Ebenfalls zur Verbesserung der Lesbarkeit haben wir die überflüssigen **1**-Operatoren der bei der Übersetzung entstehenden Präfix-Terme durch Anwendung der Axiome (3.3) und (3.4) aus Abbildung 3.4 entfernt.

6.6 Konkatenation von Diagrammen

Die Semantik der Konkatenation von Diagrammen besteht aus zwei Teilen. Zum einen müssen für jede Instanz I in jedem Diagramm D Prozeßdefinitionen in die Prozeßumgebung Θ eingefügt werden, die jeweils das Verhalten von I in D beschreiben. Das Erzeugen dieser Definitionen wird von der Funktion $[\![.]\!]^D_{instance}$ in Abbildung 6.2 durchgeführt, die die Prozeßdefinitionen $D\_I\_continue$ und die zugehörigen Parameterlisten generiert. Weiterhin müssen die Konkatenationsoperatoren in die entsprechenden Prozeßaufrufe übersetzt werden; dies geschieht mit der Funktion $[\![.]\!]^{D,I,\mathcal{E}}_{continue}$ aus Abbildung 6.11.

Wir unterscheiden bei der Übersetzung nach bedingter und unbedingter Konkatenation. Handelt es sich um eine unbedingte Konkatenation der Form

$$\textbf{continue with } D \textbf{ parameters } I_1(E_{11}, ..., E_{1n_1}), ..., I_k(E_{k1}, ..., E_{kn_k}),$$

wird der Operator im oberen Teil von Abbildung 6.11 in den Aufruf

$$D\_I\_continue(x_1, ..., x_l, E'_{i1}, ..., E'_{in_i})$$

$[\![$**continue with** D **parameters** $I_1(E_{11},...,E_{1n_1}),...,I_k(E_{k1},...,E_{kn_k})]\!]^{D,I,\mathcal{E}}_{continue} =$
$D\_I\_continue(x_{j1},...,x_{jl},[\![E_{i1}]\!]^{\mathcal{E}}_{expr},...,[\![E_{in_i}]\!]^{\mathcal{E}}_{expr})$

$[\![$**continue depending on** I'
if E_1 **then** D_1
parameters $I_1(E_{111},...,E_{11n_1}),...,I_k(E_{1k1},...,E_{1kn_k})$
...
if E_h **then** D_h
parameters $I_1(E_{h11},...,E_{h1n_1}),...,I_k(E_{hk1},...,E_{hkn_k})]\!]^{D,I,\mathcal{E}}_{continue} =$

if $I = I'$ then
$([\![E_1]\!]^{\mathcal{E}}_{expr}];$
$\_cont_{J_1}!1;...;\_cont_{J_p}!1;$
$D_1\_I\_continue(x_{j1},...,x_{jl},[\![E_{1i1}]\!]^{\mathcal{E}}_{expr},...,[\![E_{1in_i}]\!]^{\mathcal{E}}_{expr})$
$+...$
$+[[\![E_h]\!]^{\mathcal{E}}_{expr}];$
$\_cont_{J_1}!h;...;\_cont_{J_p}!h;$
$D_h\_I\_continue(x_{j1},...,x_{jl},[\![E_{hi1}]\!]^{\mathcal{E}}_{expr},...,[\![E_{hin_i}]\!]^{\mathcal{E}}_{expr}))$
else
$\_cont_I?\_choice;($
$[\_choice = 1];D_1\_I\_continue(x_{j1},...,x_{jl},[\![E_{1i1}]\!]^{\mathcal{E}}_{expr},...,[\![E_{1in_i}]\!]^{\mathcal{E}}_{expr})$
$+...$
$+[\_choice = h];D_h\_I\_continue(x_{j1},...,x_{jl},[\![E_{hi1}]\!]^{\mathcal{E}}_{expr},...,[\![E_{hin_i}]\!]^{\mathcal{E}}_{expr}))$

mit $\textsc{Inst} = \{I_1,...,I_k,\mathbf{env}\}$,
$gates = gates(I_1) \uplus ... \uplus gates(I_k) \uplus gates(\mathbf{env}) = \{m_1,...,m_l\}$,
$names = \{name(chan(m_i,J)) \mid m_i \in \textsc{Gates}, m_i \in gates(J)\}$,
$names\_vect = make\_vect(names) = x_{j1},...,x_{jl}$,
$\{J_1,...,J_p\} = \textsc{Inst} \setminus \{I',\mathbf{env}\}$

Abbildung 6.11: Übersetzungsfunktion für Konkatenation.

übersetzt. Hierbei ist E'_{ij} jeweils die Semantik des Ausdrucks E_{ij}; an die Bezeichner x_1 bis x_l sind die initialen Kanäle der Instanzen und der Umgebung gebunden. Wir nehmen an, daß I_i die zu übersetzende Instanz I ist. Im Falle einer bedingten Konkatenation der Form

continue depending on I'
  if E_1 **then** D_1 **parameters** $I_1(E_{111}, ..., E_{11n_1}), ..., I_k(E_{1k1}, ..., E_{1kn_k})$
  ...
  if E_h **then** D_h **parameters** $I_1(E_{h11}, ..., E_{h1n_1}), ..., I_k(E_{hk1}, ..., E_{hkn_k})$

müssen wir unterscheiden, ob die zu übersetzende Instanz I die Kontrollinstanz I' ist, die die Entscheidung bezüglich der Fortführung der Systemauswertung trifft. Ist die zu übersetzende Instanz I die Kontrollinstanz I', konstruiert die Übersetzungsfunktion einen Term, der eine Auswahl zwischen den h Möglichkeiten realisiert. Am Anfang der g-ten Alternative ($1 \leq g \leq h$) steht ein Testoperator, der die Gültigkeit von E_g überprüft. Ist die Bedingung erfüllt, wird die Nummer g für die getroffene Entscheidung an alle anderen beteiligten Instanzen J_1 bis J_p über die zugehörigen Kanäle $\_cont_1$ bis $\_cont_p$ gesendet. Anschließend folgt der Prozeßaufruf für den Beginn von I im Diagramm D_g ($1 \leq g \leq h$), wobei die Parameter $E'_{gi1}, ..., E'_{gin_i}$, die aus der Übersetzung von $E_{gi1}, ..., E_{gin_i}$ entstanden sind, sowie der Vektor $x_{j1}, ..., x_{jl}$ der Bezeichner, an die die initialen Kanäle gebunden sind, übergeben werden. Wir nehmen an, daß I der Instanz I_i entspricht.

Ist die zu übersetzende Instanz I nicht die Kontrollinstanz I', wird von der Übersetzungsfunktion ein Term erzeugt, in dem zuerst über den Kanal $\_cont_I$ eine Zahl empfangen wird, die die von der Kontrollinstanz ausgewählte Alternative angibt. Diese Zahl wird an den Bezeichner *_choice* gebunden. Dann wird in einer Auswahl mittels Testoperatoren die entsprechende Alternative ausgesucht und die entsprechende Prozeßdefinition mit den Kanalparametern und den Datenparametern aufgerufen.

Beispiel 6.10 (Sortierserver) *Als Beispiel für die Semantik eines Dokuments, das bedingte Konkatenation enthält, verwenden wir das Beispiel eines Servers für Sortierfunktionen. Das System besteht aus einem Server und mehreren Clients, die mit dem Server interagieren. Der Server bietet Algorithmen zum Sortieren numerischer Listen an, die von den Clients verwendet werden können. Die Verwaltung der Algorithmen auf einem Server bietet z.B. den Vorteil der einfacheren Austauschbarkeit eines Sortierverfahrens gegen eine optimierte Version, da dann nur an einer Stelle im System die Änderung vorgenommen werden muß. Hierbei haben die Clients die Auswahl zwischen zwei Möglichkeiten: Entweder sie fordern vom Server den gewünschten Algorithmus in Form einer Funktion höherer Ordnung an und führen die Sortierung selbst durch, oder sie senden die Liste der zu sortierenden Werte an den Server, der dann die Sortierung vornimmt und*

die sortierte Liste zurücksendet. Auf diese Weise können die Clients individuell entscheiden, wo die Sortierung vorgenommen werden soll (ist der Server ein wesentlich schnellerer Rechner als die Clients, ist die Sortierung auf dem Server sinnvoll; sortiert ein Client mehrere Listen mit demselben Verfahren, kann die einmalige Anforderung der entsprechenden Funktion sinnvoller sein).

In unserem Beispiel-Dokument in Abbildung 6.12 und Abbildung 6.13 spezifizieren wir ein System, das aus dem Server und einem Client besteht. Die Interaktion zwischen den anderen Clients und dem Server geschieht analog. Im initialen Diagramm Start *werden die Tore der Instanz* Server *deklariert. Über das Tor* sort_choice *teilt der Client dem Server mit, ob er die Sortierung selbst vornehmen möchte (der Wert true steht für das Sortieren beim Client). Über* get_function *mittels einer Anfrage eine Sortierfunktion vom Server angefordert werden. Wir nehmen hierzu an, daß den verschiedenen Sortierverfahren zur Identifizierung Indexnummern zugeordnet sind. Über* sort *kann der Client Sortieraufträge an den Server schicken.*

Der Client definiert zuerst eine Liste values *der zu sortierenden Zahlen. Dann entscheidet er, ob er die Sortierung selbst durchführen will und bindet den entsprechenden Wahrheitswert an* b. *Weiterhin wird an* index *die Nummer des entsprechenden Sortierverfahrens gebunden. Im Server wird die Liste* alg_list *mit den Funktionen der einzelnen Sortieralgorithmen definiert. Die Funktion* get_alg *wird zum Herausnehmen einer Funktion aus dieser Liste verwendet. Der Client sendet nun seine Entscheidung, ob er das Sortieren selbst vornehmen will, an den Server. In Abhängigkeit von dieser Nachricht entscheidet der Server, ob die Systemausführung mit dem Diagramm* Deliver_Function *oder mit* Sort_on_Server *fortgesetzt werden soll. Dabei werden im Konkatenationsoperator in beiden Fällen die Parameter* values *und* index *an den Client sowie* alg_list *und* get_alg *an den Server übergeben.*

Die Instanzdeklarationen der Diagramme Deliver_Function *und* Sort_on_Server *enthalten neben den Deklarationen der Tore auch die Parameter, die bei der Konkatenation übergeben werden. Wir haben in beiden Diagrammen dieselben Bezeichner wie im initialen Diagramm* Start *gewählt. Im Diagramm* Deliver_Function *sendet der Client eine Anfrage* get_function *an den Server, in der er den Index der gewünschten Sortierfunktion übermittelt. Der Server liest diese Funktion aus der Liste der Funktionen und sendet sie in der Antwort der Anfrage an den Client. Dieser wendet die Funktion dann auf die zu sortierende Liste an. Im Diagramm* Sort_on_Server *startet der Client eine Anfrage über das Tor* sort *mit der zu sortierenden Liste und dem Index der Sortierfunktion als Parameter. Der Server führt die Sortierung durch und sendet die sortierte Liste an den Client zurück. In beiden Diagrammen ist nun die sortierte Liste an den Bezeichner* new *gebunden.*

Im folgenden geben wir die Semantik für die Diagramme in Abbildung 6.12 und Abbildung 6.13 an. Die Prozeßumgebung Θ *enthält sechs Prozeßdefinitionen für die Semantik der beiden Instanzen in den drei Diagrammen. Im initialen Prozeßterm werden die initialen Kanäle erzeugt (da keine* while*-Schleifen verwendet*

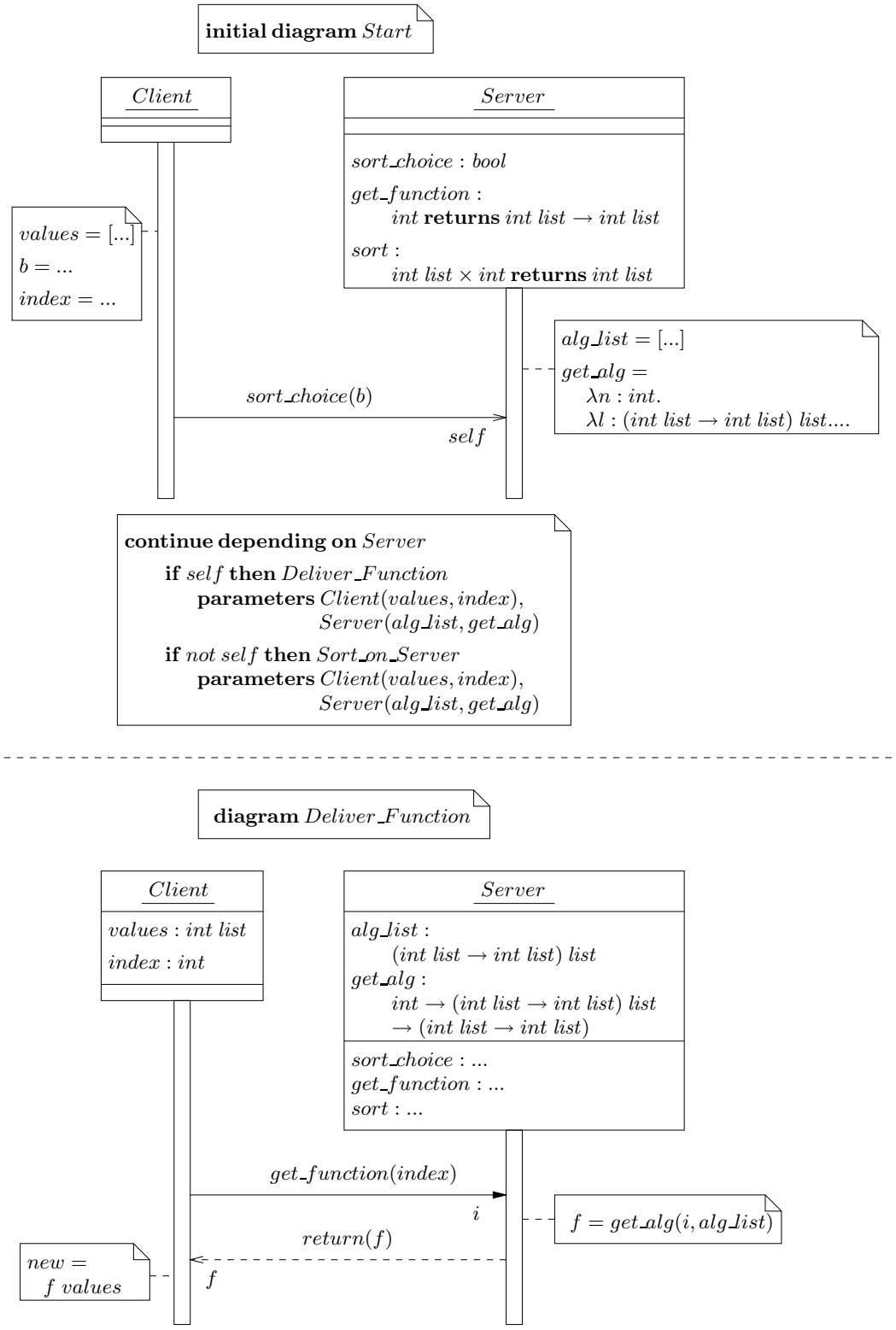


Abbildung 6.12: Beispiel für Konkatenation.

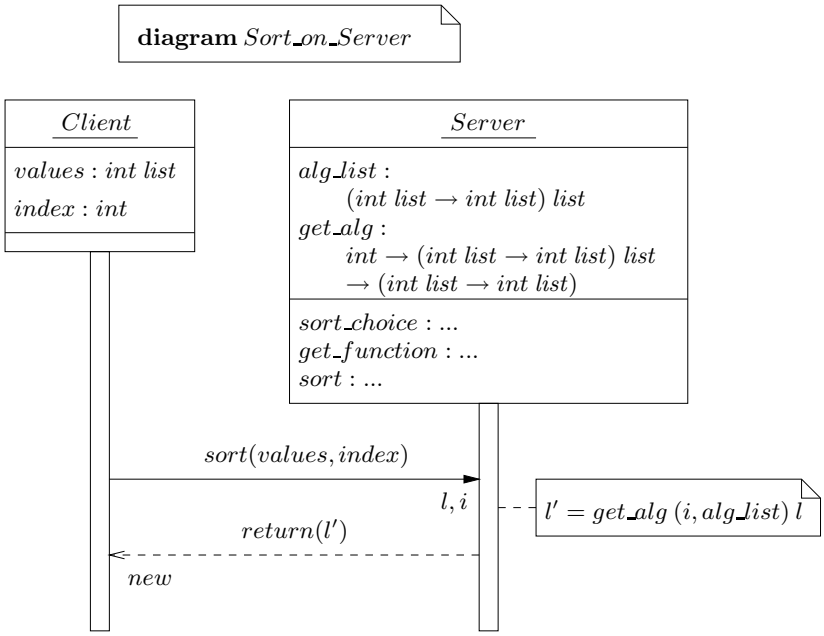


Abbildung 6.13: Beispiel für Konkatenation (Fortsetzung).

werden, geben wir hier und in den Parameterlisten der Prozeßdefinitionen die Kanäle $\_while_I$ *nicht an; siehe Axiom (3.16) in Abbildung 3.4). Anschließend werden die beiden Prozesse Start_Client_start und Start_Server_start aufgerufen; sie erhalten als Parameter die Vektoren der initialen Kanäle. Am Ende der beiden Prozesse wird die Entscheidung über die Fortsetzung der Ausführung getroffen. Der Server führt eine Fallunterscheidung anhand des Wertes von* self *durch und benachrichtigt den Client über den Kanal* $\_cont_{Client}$*. Anschließend wird der entsprechende Prozeß zur weiteren Ausführung des Servers aufgerufen. Der Client führt eine Fallunterscheidung anhand des vom Server empfangenen Werts* _choice *durch und ruft ebenfalls den entsprechenden Prozeß zur weiteren Ausführung auf. In beiden Instanzen werden neben den Kanalwerten auch die Datenparameter übergeben, die im Konkatenationsoperator angegeben sind.*

Da die Tornamen der Instanzen disjunkt sind, nehmen wir $chan(m, I) = m_{\$T\$}$ *für jedes Tor* m *der jeweiligen Instanz* I *an (*$I \in \{Client, Server\}$*).*

Prozeßumgebung Θ*:*

$Start\_Client\_start($
$\quad get\_function : (int \times (int\ list \to int\ list)\ channel)\ channel,$
$\quad sort : (int\ list \times int \times int\ list\ channel)\ channel, sort\_choice : bool\ channel,$
$\quad \_cont_{Client} : int\ channel, \_cont_{Server} : int\ channel$
$) \mapsto$

$\{values \leftarrow [...]\};$
$\{b \leftarrow ...\};$
$\{index \leftarrow ...\};$
$sort\_choice!b;$
$\_cont_{Client}?\_choice;$
$([\_choice = 1];$
$\quad Deliver\_Function\_Client\_continue(get\_function, sort, sort\_choice,$
$\quad\quad \_cont_{Client}, \_cont_{Server}, values, index)$
$+[\_choice = 2];$
$\quad Sort\_on\_Server\_Client\_continue(get\_function, sort, sort\_choice,$
$\quad\quad \_cont_{Client}, \_cont_{Server}, values, index))$

$Deliver\_Function\_Client\_continue($
$\quad get\_function : (int \times (int\ list \to int\ list)\ channel)\ channel,$
$\quad sort : (int\ list \times int \times int\ list\ channel)\ channel, sort\_choice : bool\ channel,$
$\quad \_cont_{Client} : int\ channel, \_cont_{Server} : int\ channel,$
$\quad values : int\ list, index : int$
$) \mapsto$
$\quad \textbf{ch}\ exit.$
$\quad\quad get\_function!index, exit;$
$\quad\quad exit?f;$
$\quad\quad \{new \leftarrow f\ values\}; \mathbf{1}$

$Sort\_on\_Server\_Client\_continue($
$\quad get\_function : (int \times (int\ list \to int\ list)\ channel)\ channel,$
$\quad sort : (int\ list \times int \times int\ list\ channel)\ channel, sort\_choice : bool\ channel,$
$\quad \_cont_{Client} : int\ channel, \_cont_{Server} : int\ channel,$
$\quad values : int\ list, index : int$
$) \mapsto$
$\quad \textbf{ch}\ exit.$
$\quad\quad sort!values, index, exit;$
$\quad\quad exit?new; \mathbf{1}$

$Start\_Server\_start($
$\quad get\_function : (int \times (int\ list \to int\ list)\ channel)\ channel,$
$\quad sort : (int\ list \times int \times int\ list\ channel)\ channel, sort\_choice : bool\ channel,$
$\quad \_cont_{Client} : int\ channel, \_cont_{Server} : int\ channel$
$) \mapsto$
$\quad \{alg\_list \leftarrow [...]\};$
$\quad \{get\_alg \leftarrow \lambda n : int.\lambda l : (int\ list \to int\ list)\ list....\};$
$\quad sort\_choice?self;$
$\quad ([self];$
$\quad\quad \_cont_{Client}!1;$

$Deliver\_Function\_Server\_continue(get\_function, sort, sort\_choice,$
$\quad \_cont_{Client}, \_cont_{Server}, alg\_list, get\_alg)$
$+[not\ self];$
$\quad \_cont_{Client}!2;$
$\quad Sort\_on\_Server\_Server\_continue(get\_function, sort, sort\_choice,$
$\quad \_cont_{Client}, \_cont_{Server}, alg\_list, get\_alg)$

$Deliver\_Function\_Server\_continue($
$\quad get\_function : (int \times (int\ list \to int\ list)\ channel)\ channel,$
$\quad sort : (int\ list \times int \times int\ list\ channel)\ channel, sort\_choice : bool\ channel,$
$\quad \_cont_{Client} : int\ channel, \_cont_{Server} : int\ channel,$
$\quad alg\_list : (int\ list \to int\ list)\ list,$
$\quad get\_alg : int \to (int\ list \to int\ list)\ list \to (int\ list \to int\ list)$
$) \mapsto$
$\quad get\_function?\_value;$
$\quad \{i \leftarrow \#1\ \_value, exit_{Client} \leftarrow \#2\ \_value\};$
$\quad \{f \leftarrow get\_alg(i, alg\_list)\};$
$\quad exit_{Client}!f$

$Sort\_on\_Server\_Server\_continue($
$\quad get\_function : (int \times (int\ list \to int\ list)\ channel)\ channel,$
$\quad sort : (int\ list \times int \times int\ list\ channel)\ channel, sort\_choice : bool\ channel,$
$\quad \_cont_{Client} : int\ channel, \_cont_{Server} : int\ channel,$
$\quad alg\_list : (int\ list \to int\ list)\ list,$
$\quad get\_alg : int \to (int\ list \to int\ list)\ list \to (int\ list \to int\ list)$
$) \mapsto$
$\quad sort?\_value;$
$\quad \{l \leftarrow \#1\ \_value, i \leftarrow \#2\ \_value, exit_{Client} \leftarrow \#3\ \_value\};$
$\quad \{l' \leftarrow get\_alg\ (i, alg\_list)\ l\};$
$\quad exit_{Client}!l'$

Initialer Prozeßterm:

ch $get\_function_{\$int \times (int\ list \to int\ list)\ channel\$}, sort_{\$int\ list \times int \times int\ list\ channel\$},$
$\quad sort\_choice_{\$bool\$}, \_cont_{Client\$int\$}, \_cont_{Server\$int\$}.$
$spawn(Start\_Client\_start($
$\quad get\_function_{\$int \times (int\ list \to int\ list)\ channel\$},$
$\quad sort_{\$int\ list \times int \times int\ list\ channel\$},$
$\quad sort\_choice_{\$bool\$}, \_cont_{Client\$int\$}, \_cont_{Server\$int\$}));$
$spawn(Start\_Server\_start($
$\quad get\_function_{\$int \times (int\ list \to int\ list)\ channel\$},$
$\quad sort_{\$int\ list \times int \times int\ list\ channel\$},$

$$sort\_choice_{\$bool\$}, \_cont_{Client\$int\$}, \_cont_{Server\$int\$}))$$

Die bei der Übersetzung von Bindungsoperatoren entstehenden überflüssigen **1***-Terme wurden aus Gründen der Lesbarkeit aus den Prozeßdefinitionen entfernt (siehe die Axiome (3.3) und (3.4) in Abbildung 3.4).*

6.7 Einschränkungen

Die oben angegebenen Übersetzungsfunktionen können nicht allen n-Agenten-Diagrammen, wie sie in Kapitel 5 beschrieben werden, eine eindeutige Semantik zuordnen. Im folgenden beschreiben wir die Einschränkungen der Semantik.

Race Conditions. In den n-Agenten-Diagrammen können sowohl synchrone als auch asynchrone Nachrichten auftreten. Während bei synchronen Nachrichten die Sende- und Empfangsaktion gemeinsam durchgeführt werden müssen, können diese Ereignisse bei asynchronen Nachrichten zu verschiedenen Zeitpunkten stattfinden. Daher ist es auch möglich, daß eine Instanz nacheinander mehrere Nachrichten an eine andere Instanz über dasselbe Tor sendet, so daß zu einem Zeitpunkt mehrere Nachrichten an ein Tor unterwegs sind.

In Abbildung 6.14 ist ein Diagramm mit drei Instanzen angegeben. Zuerst sendet die Instanz I_1 zweimal asynchron die Nachricht m_1 mit unterschiedlichen Parametern an die Instanz I_2. Das Senden der Nachrichten wird in den Term $spawn(m_1!1); spawn(m_1!2)$ übersetzt, der das asynchrone Senden durch das Erzeugen zweier Prozesse realisiert. Die Semantik des Empfangens der Nachrichten in I_2 lautet $m_1?\_value; \{x \leftarrow \#1\ \_value\}; m_1?\_value; \{y \leftarrow \#1\ \_value\}; \mathbf{1}$. Durch die parallele Ausführung der beiden Sendeaktionen ist die Reihenfolge der Synchronisation zwischen Sendeaktionen und Empfangsaktionen nicht eindeutig festgelegt. Synchronisiert der Prozeß für I_2 zuerst mit dem Prozeß $spawn(m_1!1)$, erhält x den Wert 1 und y den Wert 2; wird zuerst mit $spawn(m_1!2)$ synchronisiert, erhält x entsprechend den Wert 2 und y den Wert 1. Das Verhalten des Systems ist somit nichtdeterministisch. In einer Implementierung würde dies Verhalten von externen Faktoren wie der Geschwindigkeit der einzelnen Prozesse abhängen. Man spricht in diesem Fall von einer *race condition* [AHP96]: Eine *race condition* besteht, wenn zwei Ereignisse in einem Diagramm in einer *visuellen* Ordnung zueinander stehen, es aber trotzdem gezeigt werden kann, daß sie in einer konkreten Ausführung in umgekehrter Reihenfolge auftreten können. Hierbei bezeichnen wir mit visueller Ordnung die graphische Anordnung von Ereignissen auf der Lebenslinie von Instanzen.

Besonderen Einfluß auf die *race conditions* hat das der Implementierung des Systems zugrundeliegende Kommunikationsmodell. So treten z.B. *race conditions* nicht auf, wenn die Kanäle in Form von FIFO-Schlangen implementiert sind,

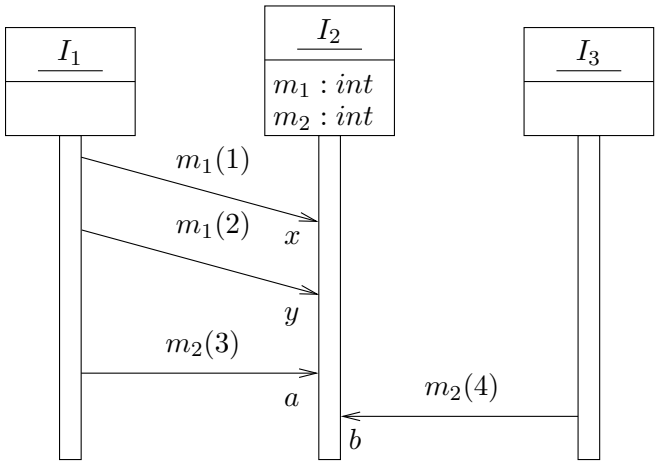


Abbildung 6.14: Diagramm ohne eindeutige Semantik.

da dann das Empfangen der Nachrichten in derselben Reihenfolge wie das Senden erfolgt. Würden die beiden Nachrichten m_1 synchron gesendet, würde keine *race conditions* bestehen, da das Empfangen der ersten Nachricht vor dem Senden der zweiten Nachricht auftreten würde. Allerdings können bei synchronen Nachrichten *race conditions* entstehen, wenn eine Nachricht nacheinander von verschiedenen Instanzen gesendet wird. In Abbildung 6.14 senden sowohl I_1 als auch I_3 die Nachricht m_2 an I_2. Die Sendeereignisse werden in der Semantik in die Terme $m_2!3$ bzw. $m_2!4$ übersetzt, die aufgrund ihrer Zugehörigkeit zu verschiedenen Instanzen parallel ausgeführt werden. Da die Terme keine Informationen über die Identität des Senders der Nachricht enthalten, entsteht auch hier eine *race condition* zwischen den beiden Nachrichten.

Auch in anderen Interaktionsdiagrammen wie den Message Sequence Charts können *race conditions* auftreten. MSCs verwenden im Unterschied zu den n-Agenten-Diagrammen ausschließlich asynchrone Kommunikation, so daß hier Abhängigkeiten wie bei der Nachricht m_1 in Abbildung 6.14 relativ häufig entstehen können. In [AHP96] wird eine Analyseverfahren vorgestellt, mit dessen Hilfe *race conditions* in MSCs gefunden werden können. Dieses Verfahren wurde auch in einem Werkzeug zur Analyse von MSCs implementiert. Aufgrund der Ähnlichkeit zu MSCs würde sich das Verfahren auch auf n-Agenten-Diagramme anwenden lassen. Allerdings müßte das Verfahren zur Erkennung von *race conditions* zwischen gleichen Nachrichten von verschiedenen Instanzen (siehe m_2 in Abbildung 6.14) modifiziert werden, da die Semantik für MSCs auch Informationen über den Absender einer Nachricht enthält [MR97, GHRW98], so daß zwischen den beiden Nachrichten m_2 in Abbildung 6.14 keine *race condition* bestehen würde.

Verschachtelung von loop-Schleifen. Die Semantik der *loop*-Schleife in Abbildung 6.9 verwendet eine Zählvariable *_count* zur Verwaltung der Anzahl der Schleifendurchläufe. Dieses Verfahren ist für nicht-verschachtelte *loop*-Schleifen ausreichend. Für die Definition der Semantik verschachtelter Schleifen ist die Verwaltung der Zählvariablen unter Vewendung eines Kellers notwendig, bei dem das jeweilige Kopfelement den Zähler für die aktuelle Schleife enthält. Dieser Keller müßte in der Datensprache nachgebildet werden.

Eine mögliche Realisierung ist die Verwendung einer Liste, bei der das Kopfelement das Kopfelement des zu simulierenden Kellers darstellt. Wird eine neue Schleifentiefe erreicht, wird ein neuer Zähler als neues Kopfelement vorne an die Liste angefügt. Nach Beendigung der Schleife wird dieses Element wieder aus der Liste entfernt. Diese Lösung erfordert aber, daß die Liste in allen beteiligten Prozessen aus Θ stets als Parameter in allen Prozeßaufrufen mitgeführt wird. Dies würde eine zusätzliche Komplexität der Übersetzungsfunktionen nach sich ziehen, die das Verständnis der wesentlicheren Konzepte erschweren würde. Daher haben wir uns bei der Definition der Semantik auf nicht-verschachtelbare *loop*-Schleifen beschränkt.

6.8 Diskussion

In diesem Kapitel haben wir eine Semantik für die n-Agenten-Diagramme aus Kapitel 5 definiert. Diese Semantik übersetzt die textuelle Notation der Diagramme in Prozeßdefinitionen des Kalküls $\mathcal{P}$ aus Kapitel 3. Wie die n-Agenten-Diagramme setzt die Semantik nicht die Verwendung einer einzelnen Datensprache voraus, sondern ermöglicht die Integration beliebiger Datensprachen, die zum einen die in Abschnitt 3.1 angegebenen Anforderungen erfüllen, zum anderen auch die Datentypen *int*, *bool* sowie Tupel mit den dazugehörigen Operationen enthalten. Wir haben als Beispiel für eine solche Integration den erweiterten λ-Kalkül aus Kapitel 4 verwendet. Auf diese Weise haben wir die erste Semantik für Interaktionsdiagramme definiert, die neben der Beschreibung des Systemverhaltens eine formale Definition der Daten beinhaltet.

Die Verwendung von $\mathcal{P}$ als Zielsprache der Übersetzungsfunktionen bietet eine Reihe von Vorteilen.

- Die Semantik von $\mathcal{P}$ erlaubt die Integration der Reduktionssemantik der gewählten Datensprache, so daß Daten- und Verhaltensaspekte in einer einheitlichen Semantik dargestellt werden können. Durch die Übersetzung der n-Agenten-Diagramme in $\mathcal{P}$ wird daher auch für die Diagramme eine einheitliche Semantik für die Daten- und Verhaltensaspekte ermöglicht. Trotzdem sind die bei der Übersetzung erzeugten Prozeßdefinitionen aufgrund ihrer syntaktischen Struktur für den Leser nachvollziehbar; in einer Semantik auf Basis des π-Kalküls [MPW92], in der Daten ebenfalls durch Prozeßterme und Kommunikation dargestellt werden müßten, wäre die Lesbarkeit

aufgrund des hohen Komplexitätsgrads wesentlich stärker eingeschränkt, da die Zuordnung der Diagrammelemente an die bei der Übersetzung erzeugten Terme nicht unmittelbar ersichtlich ist (siehe z.B. die Semantik für ALGOL in [RS99]) .

- Der unäre *spawn*-Operator ermöglicht eine einfache Darstellung der asynchronen Nachrichten, indem die entsprechende Sendeaktion durch ein *spawn* parallel zu den nachfolgenden Aktionen ausgeführt wird. Durch den *spawn*-Operator kann so die Nebenläufigkeit zwischen dem Warten auf den Empfang der Nachricht und der Ausführung der auf das Senden folgenden Ereignisse direkt im Term, der die Semantik der Instanz darstellt, realisiert werden. Bei Verwendung eines binären Operators für Parallelkomposition wäre hingegen eine zusätzliche Modellierung der gesendeten Nachricht durch einen eigenen parallelen Prozeß wie in [GHRW98] notwendig (siehe auch Abschnitt 2.3). Eine andere Möglichkeit wurde im MSC-Standard [ITU96b] realisiert, in dem die Ein- und Ausgabeaktionen als zwei Aktionen modelliert werden. Durch einen zusätzlichen Zustandsoperator wird dann gewährleistet, daß eine Eingabeaktion erst dann stattfinden kann, wenn die zugehörige Ausgabeaktion durchgeführt wurde. Da der Zustandsoperator die bereits erfolgten Ausgabeaktionen in einer Menge speichert, wird die Angabe einer axiomatischen Semantik erschwert, da diese Menge stets mit berücksichtigt werden muß.

- Weiterhin erleichtern die mobilen Kanäle von $\mathcal{P}$ die Adressierung von Nachrichten. Zum einen wird jedes Tor einer Instanz durch einen eindeutigen Kanal realisiert. Zum anderen werden die Kanäle für die Antworten in Anfragen dynamisch erzeugt und beim Senden der Nachricht mit übertragen, so daß auch hier eine korrekte Adressierung der Rückantwort gewährleistet ist. Ohne diese Möglichkeit müßte implizit für jedes Anfragetor ein entsprechendes Rückgabetor definiert und in den Parameterlisten der Prozesse mitgeführt werden. Die Fähigkeit von $\mathcal{P}$ zur dynamischen Erzeugung von Kanälen ist auch für die Angabe einer Semantik für die Kanalerzeugung notwendig.

- Für den Kalkül $\mathcal{P}$ ist eine axiomatische Semantik definiert (siehe die Abbildungen 3.4, 3.5 und 3.6), die die Äquivalenz von Prozessen bzgl. starker und schwacher Bisimulation beschreibt. Auf diese Weise kann durch Umformung von semantisch äquivalenten Termen, die durch die axiomatische Semantik als gleich betrachtet werden, die Verifikation von Spezifikationen durchgeführt werden (siehe das Beispiel des RPC-Speichers in Abschnitt 4.3.2). Allerdings ist hierfür die Unterstützung durch entsprechende Werkzeuge wünschenswert.

- Die Definition der Semantik von Daten- und Verhaltensaspekten von Sy-

stemen mit Hilfe eines Formalismus erleichtert die automatische Erzeugung von Programmcode. So lassen sich für den Kalkül $\mathcal{P}_\lambda$ Übersetzungsfunktionen definieren, die die Terme des Kalküls in Programme geeigneter Programmiersprachen überführen.

6.8.1 Vergleich mit verwandten Ansätzen

Es gibt mehrere Ansätze für die Definition der Semantik von Interaktionsdiagrammen. Im folgenden vergleichen wir die in diesem Kapitel vorgestellte Semantik für die n-Agenten-Systeme mit existierenden Semantiken für ähnliche Formen von Interaktionsdiagrammen.

Message Sequence Charts

Für *Message Sequence Charts* existieren Semantiken auf der Grundlage verschiedener semantischer Modelle. Wir diskutieren zuerst die Ansätze auf der Basis von Prozeßalgebren. Anschließend gehen wir kurz auf weitere Semantiken für MSCs ein.

Der MSC-Standard ITU-T Z.120 enthält als Anhang B eine Semantik, die die textuelle Notation der MSCs in eine Prozeßalgebra übersetzt [MR94a, MR94b, ITU96b]. Diese Semantik beschränkt sich aber auf einfache MSCs (sogenannte Basis-MSCs), so daß Kontrollstrukturen wie Schleifen und Alternativen nicht berücksichtigt werden. Ebenso wird die Behandlung von Daten nicht unterstützt. Eine Erweiterung dieser Semantik für High-level MSCs wird in [MR97] vorgestellt. Eine alternative Semantik, die ebenfalls die Komposition von Diagrammen beinhaltet, geben wir in [GHRW98] an.

Die in diesem Kapitel angebene Semantik für n-Agenten-Diagramme unterscheidet sich durch die Adressierung der Instanzen von den genannten Semantiken. In unserer Semantik geschieht die Adressierung der Instanzen ausschließlich durch die Verwendung der den Toren zugeordneten Kanäle. So ist im Beispiel in Abschnitt 6.5 in der Aktion $gegner_A!w_B$ im Schiedsrichter-Prozeß durch die Verwendung des an den Bezeichner $gegner_A$ gebundenen Kanals $gegner_{A\$bool\$}$ festgelegt, daß der *Spieler_A*-Prozeß Empfänger der Nachricht ist. In den anderen Semantiken enthalten die Kommunikationsaktionen Informationen sowohl über den Sender als auch über den Empfänger der Nachricht.

Die Prozeßalgebra $PA_\mathcal{E}$ von Mauw und Reniers in [MR94a, ITU96b] verwendet Aktionen der Form $out(s, r, m, p)$ und $in(s, r, m, p)$ zur Realisierung von Sendeereignissen, wobei s die sendende Instanz, r die empfangende Instanz, m den Nachrichtennamen und p die Parameterliste darstellt. In diesem Ansatz wird keine Kommunikation verwendet, sondern das Senden und Empfangen einer Nachricht sind unabhängige Aktionen. Die Semantik eines Diagramms besteht aus

einer Auswahloperation zwischen allen möglichen Interleaving-Sequenzen dieser Aktionen. Durch die Verwendung eines speziellen Zustandsoperators wird gewährleistet, daß nur Ereignisfolgen generiert werden können, in denen das Senden einer Nachricht vor dem entsprechenden Empfangen auftritt.

Die Algebra $\mathcal{L}_{MSC}$ aus [GHRW98] verwendet Aktionen der Form $out(r, m)@s$ und $in(s, m)@r$. Die Ausgabeaktionen erzeugen eine korrespondierende Eingabeaktion in einem durch Prozesse dargestellten Nachrichten-Pool, die dann mit der Eingabeaktion im Empfängerprozess synchronisieren kann. Die Kommunikation in $\mathcal{L}_{MSC}$ verwendet den Parralleloperator der Prozeßalgebra *TCSP* [BHR84]. Die obige Aktion $gegner_A!w_B$ würde in Algebra $\mathcal{L}_{MSC}$ aus [GHRW98] durch

$$out(Spieler\_A, gegner_A)@Schiedsrichter$$

dargestellt[3]. Zur Realisierung der asynchronen Kommunikation werden sogenannte *message pools* verwendet. Hierzu merkt sich der Operator die bereits erfolgten Sendeaktionen in einer Menge und erlaubt das Auftreten einer Empfangsaktion nur dann, wenn sich die zugehörige Sendeaktion bereits in der Menge befindet. Wird eine Nachricht gesendet, wird in dem *message pool* eine zugehörige Eingabeaktion erzeugt. Die Eingabeaktion in der empfangenden Instanz kann nur ausgeführt werden, wenn der *message pool* die gleiche Aktion enthält, d.h. Empfangsaktionen müssen sich mit dem *message pool* synchronisieren. Die Eingabeaktionen im *message pool* werden dabei als eigene Prozesse dargestellt. Der Operator $\perp$ bezeichnet den leeren *message pool*. Das Übertragen einer Nachricht m von I nach J wird im folgenden Term angegeben:

$$\begin{array}{rl} & \perp \|_E out(J, m)@I; t_I \|_\emptyset in(I, m)@J; t_J \\ \xrightarrow{out(J,m)@I} & \perp \|_\emptyset in(I, m)@J \|_E \mathbf{1}; t_I \|_\emptyset in(I, m)@J; t_J \\ \xrightarrow{in(I,m)@J} & \perp \|_\emptyset \mathbf{1}; t_I \|_E \mathbf{1}; t_J \|_\emptyset \mathbf{1} \end{array}$$

Der Operator $\|_A$ ist die Parallelkomposition aus $TCSP$ [BHR84], bei der die Aktionen in der Menge A synchronisieren müssen, während die anderen Aktionen unabhängig ausgeführt werden können. Terme für die beiden Instanzen I und J werden unabhängig voneinander ausgeführt, müssen aber alle Aktionen der Menge E aller Kommunikationsereignisse mit dem *message pool* synchronisieren. Zuerst wird die Sendeaktion durchgeführt, die eine Eingabeaktion für die Nachricht in den *message pool* einfügt. Anschließend kann der Term für J sich mit dem *message pool* synchronisieren und die Eingabeaktion ausführen.

In $\mathcal{P}$ werden asynchrone Nachrichten durch ein *spawn* als Prozeß ausgeführt. Der Term für das obige Beispiel lautet in $\mathcal{P}$ (wir nehmen an, daß die Nachricht den "leeren" Wert () überträgt):

$$\mathbf{ch}\ m.\ spawn(spawn(m!()); t_I); spawn(m?\_value; t_J)$$

[3]Da $\mathcal{L}_{MSC}$ keine Daten unterstützt, wird der Bezeichner w_B nicht angegeben.

Daher kann t_I bereits Aktionen ausführen, bevor die Kommunikation über m stattgefunden hat. Hierdurch ist eine Realisierung der asynchronen Kommunikation, im Gegensatz zu den Algebren in [MR94a, MR97, GHRW98], ohne die Einführung eines speziellen Operators möglich. Dies vereinfacht die Angabe von Äquivalenzen auf Termen und die Definition einer axiomatischen Semantik.

Die Definition der Sprache *High-level MSC* von Mauw und Reniers in [MR97] erlaubt die Verknüpfung von einzelnen MSCs zu größeren MSC-Dokumenten. Wie auf Seite 162 erläutert, findet bei der Komposition von Basis-MSCs keine Synchronisation der Instanzen statt. Daher ist es möglich, daß eine Instanz mit ihrer Ausführung schon im folgenden Diagramm angekommen ist, während die Ausführung einer anderen Instanz sich noch im vorherigen Diagramm befindet. Aus dieser fehlenden Synchronisation folgt auch das Problem des *global choice*: Die Instanzen müssen "wissen", mit welchem Diagramm die Systemausführung fortgesetzt werden soll, obwohl keine Systemkomponente explizit die Auswahl trifft. Semantiken für HMSCs müssen diese Art der Komposition von Diagrammen modellieren. Die Semantik in [MR97] verwendet hierzu den Operator zur schwachen sequentiellen Komposition von Rensink und Wehrheim aus [RW94]. Dieser Operator beschränkt die zeitliche Reihenfolge auf explizit als abhängig definierte Ereignisse und erlaubt die beliebige Reihenfolge unabhängiger Ereignisse. In [MR97] werden zwei Operatoren für sequentielle Komposition verwendet: Innerhalb einer Instanz werden die einzelnen Ereignisse mittels starker sequentieller Komposition verknüpft, während MSCs durch die schwache sequentielle Komposition verbunden werden. Die hierdurch enstehende Semantik ist relativ komplex und erfordert die Einführung weiterer spezieller Operatoren wie die verzögerte Auswahl (*delayed choice*). Aus diesem Grund wurde in [GHRW98] eine alternative Semantik für die Komposition von MSCs vorgestellt. Dieser Vorschlag verwendet eine spezielle Version des Operators aus [RW94], die die explizite Abhängigkeitsrelation durch eine implizite Abhängigkeit auf Instanzen ersetzt. Hierdurch kann sowohl die sequentielle Komposition von Ereignissen als auch die von MSCs durch einen einheitlichen Operator beschrieben werden.

In den n-Agenten-Diagrammen ist nur die unbedingte Konkatenation eine schwache sequentielle Komposition von Diagrammen. Jede Instanz ruft unabhängig von den anderen den Prozeß auf, der ihre Semantik im nachfolgenden Diagramm beschreibt. Unsere Semantik benötigt hierfür keinen speziellen Operator zur Realisierung der schwachen sequentiellen Komposition, da ein vergleichbarer Effekt durch die Verbindung von starker sequentieller Komposition und dem unären *spawn*-Operator realisiert wird. Bei der bedingten Konkatenation ruft zwar auch jede Instanz für sich den jeweilgen Nachfolgeprozeß auf; zuvor interagiert sie aber mit der Kontrollinstanz, so daß hier eine Synchronisation erfolgt. Daher tritt hier das Problem des *global choice* nicht auf.

In den Semantiken für MSCs enthalten die Aktionen zur Darstellung von Sende- und Empfangsereignissen auch Informationen über den Sender einer Nach-

richt. Daher treten *race conditions* zwischen verschiedenen Instanzen (siehe die Nachrichten $m_2(3)$ und $m_2(4)$ in Abbildung 6.14) nicht auf. Hingegen sind *race conditions* bzgl. des mehrfachen Sendens einer Nachricht von einer Instanz zu einer anderen (siehe $m_1(1)$ und $m_1(2)$ in Abbildung 6.14) auch in diesen Semantiken möglich.

Neben den Semantiken, die auf Prozeßalgebren basieren, wurden weitere Semantiken für MSCs definiert. Eine frühe Semantik von MSCs auf der Basis von Petri-Netzen [Rei82] wurde von Graubmann, Rudolph und Grabowski in [GRG93] vorgestellt. Während die prozeßalgebraischen Semantiken *Interleaving*-Semantiken sind, die nebenläufiges Verhalten durch eine zeitlich verschachtelte Ausführung der Komponenten modellieren, bildet die Semantik auf der Basis von Petri-Netzen die Nebenläufigkeit der Diagrammkomponenten auf parallele Abläufe in den Petri-Netzen ab (*true concurrency*-Semantik). Da für Petri-Netze eine Reihe von Werkzeugen existieren, lassen sich auf diese Weise Analysen des *true concurrency*-Verhaltens von Systemen durchführen. Zum Beispiel lassen sich aus Petri-Netzen Halbordnungssemantiken ableiten, die für die Analyse von MSCs bzgl. *race conditions* (siehe Seite 201) geeignet sind[4]. Die Semantiken für MSCs auf der Basis von Petri-Netzen definieren nur die Bedeutung der Basis-MSCs. Die Komposition von Diagrammen wie in den High-level MSCs wird nicht unterstützt.

In [LL94a] geben Ladkin und Leue eine Semantik für MSCs an, bei der die Diagramme in Transitionssysteme (siehe Definition 2.1) übersetzt und anschließend in Büchi-Automaten [Tho90] überführt werden. Hierbei wird vorausgesetzt, daß die Ausführung von Übersetzungen von Diagrammen nur endlich viele Zustände benötigt. In [GHRW98] weisen wir darauf hin, daß diese Annahme in Systemen, die Konkatenation von MSCs enthalten, nicht immer zutreffend ist, sondern durch die Iteration asynchroner Nachrichten eine unendliche Zustandsmenge erzeugt werden kann (siehe Seite 162).

Alle oben angeführten Ansätze für Semantiken für MSCs beschränken sich auf die Definition des von den Diagrammen beschriebenen Verhaltens und abstrahieren von den Datentransformationen. Daher muß die Semantik der in Systemen auftretenden Daten in einem zusätzlichen Formalismus spezifiziert werden. Durch die fehlenden formalen Beziehungen zwischen den Teil-Semantiken für Daten- und Verhaltensaspekte wird die Verifikation von Systemen und die automatische Codeerzeugung erschwert.

Der neue Standard MSC 2000 [ITU99] enthält zwar eine Beschreibung von Daten; diese wird aber nicht formal definiert, so daß eine zu [ITU96b] vergleichbare formale Semantik für MSC 2000 noch nicht vorliegt.

[4]Ein Beispiel für eine solche Verwendung einer Halbordnungssemantik, die nicht auf Petri-Netzen basiert, ist in [AHP96] angegeben.

Interaktionsdiagramme von UML

Für die Unified Modeling Language [BRJ98, RJB98] exisitiert eine Semantik, die die Bedeutung der Sprachelemente definiert. Diese Semantik liegt in Form eines *Metamodells* vor: Die Bedeutung der Sprachelemente ist in Form von UML-Diagrammen definiert. Die Semantik legt die Beziehungen der Diagrammelemente untereinander fest. So wird z.B. festgelegt, daß eine Nachricht ein Objekt der Klasse *MessageInstance* ist, die wiederum eine Instanz einer Unterklasse von *Request* ist. Hingegen beschreibt die Semantik nicht die möglichen Systemabläufe, die mit den dynamischen Modellen spezifiziert werden. Daher ist zusätzlich zu der Semantik von UML die Definition einer weiteren Semantik notwendig, die das dynamische Systemverhalten beschreibt.

In [GGW98, GGW99] verwenden wir Petri-Netze, um die Semantik der Kollaborationsdiagramme von UML zu diskutieren. Wir verwenden sogenannte *Übersetzungsschemata*, die einzelne Konstrukte der Diagramme in Teile von Petri-Netzen übersetzen. Diese werden dann kompositionell zur Semantik eines gesamten Diagramms zusammengesetzt. Jede Instanz wird auf eine Folge von Stellen und Transitionen abgebildet. Die Transitionen repräsentieren hierbei die Aktionen, die die Instanzen ausführen (z.B. Senden und Empfangen von Nachrichten). Anhand der Übersetzungsschemata werden unterschiedliche Interpretationsmöglichkeiten für das von den Diagrammen beschriebene Verhalten diskutiert, so z.B. bzgl. des Verhaltens asynchroner Nachrichten und bzgl. Iteration. Da die Übersetzungsschemata als Grundlage für die Diskussion über mehrdeutige Sprachkonstrukte dienen sollen, wurde die Übersetzung der Diagramme in Petri-Netze nur informell ohne eine formale Definition des Übersetzungsvorgangs angegeben.

Für die Sequenzdiagramme von UML wird von Störrle in [Stö99] eine Übersetzung der Diagramme in Petri-Netze höherer Ordnung [JR91] angegeben. Da für die Diagramme von UML keine standardisierte textuelle Notation vorliegt, wird eine eigene Syntax verwendet. Für die systematische Konstruktion von Petri-Netzen werden Operatoren auf Netzen definiert, die z.B. die Transitionen für das Sende- und das Empfangsereignis einer Nachricht bzgl. synchroner oder asynchroner Kommunikation verbinden. Die Semantik unterstützt nur wenige Sprachkonstrukte der Sequenzdiagramme. So fehlen z.B. bedingte Nachrichten und Iteration.

In [Kna99] wird von Knapp eine Semantik für Interaktionsdiagramme auf der Basis von Temporaler Logik definiert. Zusätzlich wird eine Semantik auf der Basis von *pomsets* [Pra86] angegeben und es wird gezeigt, daß die erzeugten *pomsets* von den Formeln der Temporalen Logik beschrieben werden.

In [FHD+99] stellen wir eine Semantik für Sequenzdiagramme vor, die zusätzlich Echtzeitanforderungen enthalten [Dou97]. Die Sequenzdiagramme werden in *timed automata* [AD94] übersetzt. Diese bilden eine Erweiterung der endlichen Automaten um Uhren und Zeitbedingungen (siehe Abschnitt 7.3.4). Wir verwen-

den die Semantik zur Verifikation eines HiFi-Protokolls mit Hilfe des UPPAAL-Werkzeugs [LPY97].

Wie bei den Message Sequence Charts abstrahieren die bisherigen Semantiken für die Interaktionsdiagramme von UML von den Daten der spezifizierten Systeme, so daß auch hier kein Formalismus existiert, der sowohl die Semantik der Daten- als auch die Verhaltensaspekte von Systemen beschreibt. Da in UML keine Konstrukte für die Komposition von Diagrammen existieren, unterstützen die Semantiken nicht die Zerlegung einer Verhaltensbeschreibung in mehrere Diagramme.

Kapitel 7

Schlußbemerkungen und Ausblick

In diesem Kapitel fassen wir die in der Arbeit erzielten Ergebnisse noch einmal zusammen. Weiterhin erläutern wir die Unterschiede zu verwandten Ansätzen und diskutieren mögliche Inhalte weiterer Forschungsarbeiten.

7.1 Abschließende Bemerkungen

Anhand der in Abschnitt 1.2 definierten Ziele diskutieren wir die im Rahmen dieser Arbeit erzielten Ergebnisse.

Der Kalkül $\mathcal{P}$ vereinigt die Konzepte einer mobilen Prozeßalgebra mit einer funktionalen Beschreibung von Daten. Hierbei wird nicht die Verwendung einer speziellen Datensprache festgelegt, sondern es kann jede funktionale Sprache verwendet werden, die eine kleine Menge von Anforderungen bzgl. ihrer Datentypen und der Reduktionssemantik erfüllt. Auf diese Weise kann zum einen die Komplexität der Datensprache (und damit auch die der Reduktionssemantik) an das zu spezifizierende Problem angepaßt werden, zum anderen wird die flexible Auswahl von geeigneten Sprachkonstrukten ermöglicht. Die Integration der Daten geschieht durch die Verwendung von Ausdrücken zur Berechnung der Werte, die in den Operationen der Prozeßalgebra verwendet werden. Durch die Darstellung der Kommunikationskanäle als Werte der Datenteilsprache wird die geforderte Eigenschaft der Mobilität erzielt, da Kanalwerte wie die Werte anderer Datentypen in Kommunikationsaktionen gesendet und empfangen werden können. Weiterhin enthält $\mathcal{P}$ einen Operator zur dynamischen Kanalerzeugung, der dem Restriktionsoperator des π-Kalküls entspricht.

In $\mathcal{P}$ wird die Nebenläufigkeit von Prozessen, wie in Abschnitt 1.2 gefordert, durch einen unären *spawn*-Operator realisiert. Dieser Operator entspricht mehr als der binäre Operator für Parallelkomposition aus anderen Prozeßalgebren der

Realisierung von Nebenläufigkeit in parallelen Programmiersprachen oder Betriebssystemen (z.B. *fork* in *Unix*). Dies erleichtert die direkte Umsetzung von Spezifikationen in Programmcode der entsprechenden Sprachen bzw. Bibliotheken. Die Anwendung von $\mathcal{P}$ zur Definition von Semantiken in Kapitel 6 und [Geh98] hat zudem gezeigt, daß sich mit Hilfe des *spawn*-Operators asynchrone Kommunikation einfach darstellen läßt.

Zur Charakterisierung wohlgetyper Terme wurde ein Typsystem entwickelt, in das das Typsystem der jeweils gewählten Datensprache integriert werden kann. Die Bedeutung der Operationen von $\mathcal{P}$ wurde durch eine operationelle Semantik definiert, die die Ausführung von Prozessen auf Transitionssysteme abbildet. Hierbei gelang es, die Semantik von *spawn*, im Gegensatz zu verwandten Ansätzen, ohne die Verwendung eines zusätzlichen Hilfsoperators zu beschreiben. Als Äquivalenzen auf Termen wurden die starke und die schwache Bisimulation definiert und an die Behandlung von Daten angepaßt: Im Gegensatz zur Standard-Bisimulation [Mil89, MPW92] müssen die Transitionsbeschriftungen nicht identisch, sondern bzgl. einer Äquivalenzrelation auf Daten gleich sein. Da die schwache Bisimulation keine Kongruenz für den Auswahloperator ist (vergleiche [Mil89]), wurde zusätzlich die schwache Kongruenzrelation definiert. Es wurde gezeigt, daß die starke Bisimulation und die schwache Kongruenz Kongruenzrelationen für alle Operatoren von $\mathcal{P}$ sind. Die Relationen wurden durch die Angabe einer axiomatischen Semantik charakterisiert, die die Äquivalenz von Termen durch Gleichheitsregeln beschreibt. Es wurde die Korrektheit der axiomatischen Semantik bzgl. der genannten Äquivalenzrelationen bewiesen. Die Vollständigkeit des axiomatischen Systems wurde hingegen nicht untersucht, da der zu führende Beweis sehr aufwendig wäre und das Vollständigkeitsresultat für die praktische Einsetzbarkeit der axiomatischen Semantik nur eine geringe Bedeutung hat.

Als Beispiel für die Integration einer Datensprache in $\mathcal{P}$ haben wir einen erweiterten, getypten λ-Kalkül verwendet; den Einsatz des so entstandenen Kalküls $\mathcal{P}_\lambda$ haben wir anhand eines Referenzbeispiels [BMS96] für nebenläufige Systeme demonstriert. Nach erfolgter Modellierung des Systems wurde gezeigt, daß zwei Varianten des Systems äquivalentes Verhalten bzgl. der schwachen Kongruenz zeigen. Hierzu wurden die Axiome der axiomatischen Semantik für die Umformung von Termen verwendet.

Zur graphischen Spezifikation von verteilten reaktiven Systemen mit Daten wurden die n-Agenten-Diagramme eingeführt. Diese Diagramme verbinden die aus den Message Sequence Charts [ITU96a, ITU99] und den Sequenzdiagrammen der *Unified Modeling Language* [BRJ98] bekannte Beschreibung des Systemverhaltens durch graphische Darstellung des Nachrichtenflusses mit der Beschreibung der beim Systemablauf vorgenommenen Datentransformationen. Im Gegensatz zu den MSCs und den dynamischen Modellen von UML gehen die n-Agenten-Diagramme von einem konkreten Systemmodell aus, bei dem ein System aus einer festen Anzahl verteilter Komponenten mit jeweils lokalem Speicher besteht.

Durch die Vorgabe eines Systemmodells ist die Eindeutigkeit der Semantik der Sprachkonstrukte gegeben, andererseits bedeutet dies eine Einschränkung in der Ausdrucksfähigkeit der Sprache. Allerdings lassen sich auch prozedurale Systeme mit den n-Agenten-Diagrammen spezifizieren, sofern auf die Verwendung globaler Daten verzichtet wird. So lassen sich z.B. Prozeduraufrufe durch Anfragen darstellen.

Wie bei dem $\mathcal{P}$-Kalkül ist die Datenspache nicht vorgegeben, sondern es kann jede funktionale Sprache verwendet werden, die Minimalanforderungen bzgl. der Datentypen und der zugehörigen Operatoren erfüllt und die über eine Reduktionssemantik verfügt. Daher läßt sich auch hier eine problemorientierte Datenbehandlung realisieren.

Die Diagramme verwenden die Syntax der Sequenzdiagramme aus der *Unified Modeling Language* (UML) [BRJ98, RJB98], da die graphischen Elemente von UML aufgrund der breiten Akzeptanz dieser Sprache zunehmend von Entwicklungstools und graphischen Werkzeugen unterstützt werden. Die Bedeutung der Diagrammelemente ist aber im Gegensatz zu der der Sequenzdiagramme und der *Message Sequence Charts* [ITU96a, ITU99] speziell auf die Spezifikation verteilter Systeme hin ausgerichtet. So wird zwischen synchroner und asynchroner Kommunikation unterschieden, wobei bei synchroner Kommunikation Sende- und Empfangsereignis einer Nachricht gleichzeitig ausgeführt werden müssen, während sie bei asynchroner Kommunikation getrennt auftreten dürfen. Weiterhin können Anfragen (synchron oder asynchron) spezifiziert werden, bei denen der Sender auf eine Rückantwort vom Empfänger warten muß. Neben der graphischen Syntax wurde eine textuelle Notation definiert, die die textuelle Darstellung der Diagramme erlaubt.

Die n-Agenten-Diagramme unterstützen die Erstellung generativer Spezifikationen durch komplexe Sprachelemente wie z.B. Schleifen. Es ist möglich, die Beschreibung von Systemabläufen aus einzelnen Diagrammen durch Komposition zusammenzusetzen. Entscheidungen über die Auswahl einer Alternative oder die Fortführung von Schleifen werden in Abhängigkeit von lokalen Daten getroffen. Sind an einem komplexen Sprachkonstrukt mehrere Instanzen beteiligt, fungiert eine der Instanzen als Kontrollinstanz, die die entsprechenden Entscheidungen fällt. Hierdurch werden semantische Mehrdeutigkeiten vermieden.

Zur Definition der formalen Semantik der n-Agenten-Diagramme werden Übersetzungsfunktionen definiert, die, ausgehend von der textuellen Darstellung, die Diagramme in Prozeßdefinitionen des $\mathcal{P}$-Kalküls übersetzen. Diese Semantik wird dann mit der Reduktionssemantik der gewählten funktionalen Datensprache parametrisiert, so daß eine integrierte formale Semantik der n-Agenten-Diagramme entsteht. Diese Semantik kann zum einen analysiert werden (z.B. durch Anwendung der axiomatischen Semantik von $\mathcal{P}$), zum anderen ist die direkte Umsetzung in eine parallele Programmiersprache möglich. Weiterhin hat die Definition der Semantik der n-Agenten-Diagramme die Einsetzbarkeit des Kalküls $\mathcal{P}$ als semantische Basis von parallelen Sprachen bzw. Notationen mit Daten gezeigt.

7.2 Vergleich mit verwandten Ansätzen

In diesem Abschnitt führen wir einen abschließenden Vergleich mit verwandten Ansätzen durch.

7.2.1 Prozeßkalküle

Prozeßalgebren lassen sich in die Klassen der Kalküle ohne bzw. mit Mobilität einteilen. Die bekanntesten Kalküle der ersten sind *CCS* von Milner [Mil89], *CSP* von Hoare [Hoa85], *TCSP* von Brookes, Hoare und Roscoe [BHR84] sowie *ACP* von Baeten und Weijland [BW90]. In diese Klasse gehören auch APC von Baeten und Vaandrager [BV92], μCML von Ferreira, Hennessy und Jeffrey [FHJ95] sowie der *Fork Calculus* von Havelund und Larsen [HL94]. Diese Sprachen betrachten die Kommunikationsstruktur als statisch, so daß zur Laufzeit eines Systems keine Änderungen der Systemstruktur möglich sind. Der bekannteste Kalkül aus der Klasse mit Mobilität ist der π-Kalkül von Milner, Parrow und Walker [MPW92], der das Übertragen von Kanalnamen in Kommunikationsaktionen ermöglicht. Verwandte Kalküle sind der *Fusion Calculus* von Parrow und Victor [PV98] sowie der ψ-Kalkül von Havelund [Hav94].

In Abbildung 7.1 geben wir eine Übersicht über die Merkmale einiger Prozeßkalküle. In den Spalten der Tabelle sind die einzelnen Merkmale, nach denen wir unterscheiden, aufgelistet. Wir erläutern im folgenden die Abkürzungen für die Ausprägungen der Merkmale.

Die Spalte *Hintereinanderausführung* beschreibt, auf welche Weise Aktionen sequentiell ausgeführt werden. Hierbei bezeichnet A die Verwendung von Aktionspräfixen und S die Verknüpfung durch sequentielle Komposition. S/A steht für die Verwendung von Aktionspräfixen bei Operationen, die Bindungen erzeugen, und für sequentielle Komposition bei allen anderen Operatoren. Die Spalte *Parallelität* gibt mittels PK bzw. SP an, ob die Nebenläufigkeit durch einen binären Parallelоperator oder durch einen unären *spawn*-Operator realisiert wird. Mit (SP) geben wir an, daß der Operator zur Prozeßerzeugung in der Semantik durch einen binären parallelen Hilfsoperator definiert wird. In der Spalte *Kommunikation* wird das Kommunikationsmodell des jeweiligen Kalküls angegeben. Hier steht CCS für das Modell von CCS, bei dem jeweils eine Eingabe- und eine Ausgabeaktion kommunizieren. Die Synchronisation hängt nur vom Kanalnamen ab und wird nicht durch den Parallelоperator beeinflußt. Kommunikationsaktionen können auch unabhängig ausgeführt werden, falls die Kommunikation nicht durch Restriktion erzwungen wird. Bei Kommunikation wie in TCSP enthält der Parallelоperator $\|_A$ eine Menge A der Kanäle, über die kommuniziert werden muß. Aktionen, die andere Kanäle verwenden, können unabhängig ausgeführt werden. In diesem Modell ist auch die Kommunikation von mehr als zwei Prozessen innerhalb einer Aktion möglich. Bei der ACP-Kommunikation wird der Parallelоperator durch eine Kommunikationsfunktion γ ergänzt: $\gamma(a, b) = c$ besagt, daß, wenn

	Hintereinander-ausführung	Parallelität	Kommunikation	Mobilität	I/O-Aktionen	Daten	Axiomatische Semantik
CCS	A	PK	CCS	-	$=$	-	+
TCSP	A	PK	TCSP	-	$=$	-	+
ACP	S	PK	ACP	-	$=$	-	+
APC	S	(SP)	ACP	-	$=$	-	+
$Fork$-Kalkül	S	(SP)	CCS	-	$=$	-	+
LOTOS	S/A	PK	TCSP	-	$\neq$	AD	+
E-ELOTOS	S/A	PK	TCSP	-	$\neq$	(FP)	-
μCML	S	(SP)	CCS	-	$\neq$	(FP)	-
π-Kalkül	A	PK	CCS	+	$\neq$	KN	+
$Fusion$-Kalkül	A	PK	CCS	+	$=$	KN	+
ψ-Kalkül	S/A	(SP)	CCS	+	$\neq$	KN	-
$\mathcal{P}$-Kalkül	S/A	SP	CCS	+	$\neq$	FP	+

Abbildung 7.1: Merkmale von Prozeßkalkülen.

ein Prozeß eine Aktion a und ein anderer eine Aktion b durchführen kann, können beide Prozesse gemeinsam eine c-Aktion ausführen. In der Spalte *Mobilität* geben wir an, ob der jeweilige Kalkül das Konzept der Mobilität unterstützt. Die Spalte $I/O-Aktionen$ gibt an, ob Eingabe- und Ausgabeaktionen in der Semantik gleich oder unterschiedlich behandelt werden. Eine unterschiedliche Behandlung liegt in den Kalkülen vor, in denen die Eingabeaktion im Gegensatz zur Ausgabeaktion Bindungen erzeugt. Die Spalte *Daten* enthält Informationen darüber, ob und in welcher Form der Kalkül die Behandlung von Daten erlaubt. Hierbei steht $-$ für fehlende Datenunterstützung, AD für abstrakte Datentypen, KN für Kanalnamen und FP für funktionale Programmierung. Mit (FP) bezeichnen wir Kalküle, die zwar Elemente der funktionalen Programmierung für die Datenbeschreibung verwenden, denen aber gebräuchliche Sprachkonzepte wie z.B. Funktionen höherer Ordnung [Thi94] fehlen. Abschließend geben wir in der Spalte *Axiomatische Semantik* an, ob für den jeweiligen Kalkül eine axiomatische Semantik existiert.

Anhand von Abbildung 7.1 vergleichen wir den $\mathcal{P}$-Kalkül aus Kapitel 3 mit den anderen angeführten Prozeßalgebren.

Hintereinanderausführung. Die Verwendung von sequentieller Komposition zur Hintereinanderausführung von Aktionen hat gegenüber der Verwendung von Aktionspräfixen eine höhere Ausdrucksfähigkeit der Sprache zur Folge. Allerdings ergibt sich bei der Anwendung von Substitutionen das Problem des Gültigkeitsbereichs, in dem die Substitution eines Bezeichners erfolgen soll. Daher verwendet $\mathcal{P}$ wie die anderen Kalküle mit Datenbehandlung und der ψ-Kalkül eine eingeschränkte Form der sequentiellen Komposition, die Operationen mit Erzeugung von Bindungen als Präfixoperator realisiert. Auf diese Weise ist der Gültigkeitsbereich eines Bezeichners stets eindeutig. Uneingeschränkte sequentielle Komposition tritt bisher nur ein Algebren ohne Daten und ohne Mobilität auf.

Parallelität. $\mathcal{P}$ verwendet wie APC, *Fork*-Kalkül, μCML und ψ-Kalkül einen unären Operator zur Prozeßerzeugung. Die übrigen Prozeßalgebren verwenden einen binären Operator zur Parallelkomposition. Die Semantik von $\mathcal{P}$ erlaubt eine direkte Beschreibung der Bedeutung dieses Operators, während die Semantiken der anderen Kalküle hierzu binäre Hilfsoperatoren für sequentielle Komposition einführen. Dies kann die Definition einer axiomatischen Semantik erschweren (siehe Abschnitt 2.3 sowie [HL94]).

Kommunikation. Wie alle anderen mobilen Kalküle verwendet $\mathcal{P}$ das Kommunikationsmodell von CCS, bei dem zwei Aktionen eine gemeinsame Kommunikation ausführen und die Synchronisation ausschließlich durch den verwendeten Kanal erfolgt. Dies erleichtert die Realisierung der Mobilität, da eine Änderung in der Adressierung einer Aktion durch einfache Substitution des Kanals erzielt werden kann. Im TCSP-Modell der Kommunikation besitzt der Paralleloperator $\|_A$ eine Menge A der Kanäle, über die synchronisiert werden soll. Eine Ersetzung eines Kanals durch einen anderen müßte entsprechend alle Synchronisationsmengen anpassen, die den ersetzten Kanal enthalten. Durch diesen nicht-lokalen Effekt ist eine Substitution aufwendiger als im CCS-Modell. Ein ähnliches Problem tritt bei der Verwendung von ACP auf. Hier müßte bei einer Substitution eines Kanalnamens neben dem Prozeßterm auch die Kommunikationsfunktion γ entsprechend modifiziert werden, so daß der aktuelle "Zustand" von γ dynamisch in den Zuständen der Semantik protokolliert werden müßte.

Mobilität Während in mobilen Kalkülen wie dem π-Kalkül [MPW92] und dem *Fusion*-Kalkül [PV98] Kanalnamen sowohl als Kanalwerte als auch als Bezeichner, die durch Kanäle substituiert werden können, verwendet werden, unterscheidet $\mathcal{P}$ zwischen Bezeichnern und Werten. Hierdurch ist eine Unterscheidung von offenen und geschlossenen Termen möglich, so daß für $\mathcal{P}$ im Gegensatz zum π-Kalkül Bisimulation eine Kongruenz bzgl. der Anwendung von Substitutionen ist (siehe Seite 51).

Kanäle sind Werte eines speziellen Datentyps. Hierdurch erlaubt $\mathcal{P}$ eine weitergehende Beschreibung von Kommunikationsstrukturen als die zuvor genannten Kalküle, da Kanalwerte in Datenstrukturen abgelegt werden können. Weiterhin ist eine Unterscheidung zwischen einer monadischen und einer polyadischen Version wie beim π-Kalkül nicht notwendig, da polyadische Kommunikation durch die Übertragung von Tupeln realisiert werden kann.

Daten. Der Kalkül $\mathcal{P}$ unterscheidet sich von den anderen mobilen Prozeßalgebren durch seine Unterstützung von Datenbeschreibungen. Kanäle sind Werte der Datensprache und können somit wie die Werte anderer Datentypen in Datenstrukturen abgelegt werden. Die anderen mobilen Kalküle unterstützen nur die Verwendung von Kanalnamen, so daß Datenoperationen mit Hilfe der vorhandenen Sprachelemente nachgebildet werden müssen. Kodierungen des λ-Kalküls sind in [MPW92] und [PV98] für den π-Kalkül bzw. den *Fusion*-Kalkül angegeben. Diese Codierungen sind durch ihren geringen Abstraktionsgrad nur sehr eingeschränkt für die problemorientierte Datenbeschreibung verwendbar.

Die übrigen Kalküle mit einer über Kanalnamen hinausgehenden Beschreibung von Daten unterstützen nicht die Verwendung von Mobilität. Weiterhin enthalten die Kalküle mit einer funktionalen Datenbeschreibung nur eine Teilmenge der funktionalen Programmierung. Die Sprache E-LOTOS realisiert Funktionen als eine spezielle Form von deterministischen Prozessen, so daß Funktionen höherer Ordnung nicht unterstützt werden. Die Sprache μCML [FHJ95], die für die Definition der Semantik des Sprachkerns von *Concurrent ML* [Ren92, Rep99] entworfen wurde, verwendet nur rudimentäre Funktionen für ganzzahlige Arithmetik. Die Spezifikation von Daten mit Hilfe abstrakter Datentypen wurde bisher nur in Prozeßalgebren ohne Mobilität untersucht [BB87, ISO87, GP94a, GP94b].

I/O-Aktionen. Während in den Prozeßalgebren ohne Daten und ohne Mobilität die Eingabe- und Ausgabeaktionen gleich behandelt werden (es gibt i. allg. nur eine gemeinsame Transitionsregel für beide Aktionsarten), unterscheiden die Kalküle, die Bindungen erzeugen, zwischen beiden Arten von Aktionen. Der Grund hierfür ist, daß die bei Eingabeaktionen empfangenen Werte an Bezeichner gebunden werden, so daß im Gültigekeitsbereich der Bezeichner durch den gelesenen Wert bzw. Kanalnamen ersetzt werden muß. Die Ausnahme ist hierbei der *Fusion*-Kalkül, der durch sein Konzept der Namensäquivalenzen und der globalen Ersetzung von Namen die Ein- und Ausgabeaktionen gleich behandelt. Allerdings tritt hier das Phänomen der globalen Ersetzung auf, bei dem eine Substitution eines Bezeichners auch in den Teilen des Systems vorgenommen wird, die an der Kommunikationsaktion nicht beteiligt sind. Daher werden hier Gültigkeitsbereiche unabhängig von den Eingabeaktionen durch Restriktion definiert.

Äquivalenzrelationen und axiomatische Semantik. Für $\mathcal{P}$ wurden als Äquivalenzrelationen auf Termen Varianten der starken und der schwachen Bisimulation [Mil89] definiert. Diese Äquivalenzen unterscheiden sich von der Standarddefinition, indem sie nicht die Identität der Transitionsbeschriftungen fordern, sondern eine Äquivlenz bzgl. der Reduktionssemantik der Datensprache betrachten. Die Äquivalenzen wurden in Form einer axiomatischen Semantik charakterisiert.

Die Sprache μCML integriert im Gegensatz zu $\mathcal{P}$ die Verhaltens- und die Datenteilsprache symmetrisch, da die Sprachelemente für die Verhaltensbeschreibung als Operatoren der Datensprache definiert sind. Dies erfordert die Verwendung einer Bisimulation höherer Ordnung, da auch die Transitionsbeschriftungen bisimulär sein müssen [FHJ95]. Für diese komplexe Äquivalenzrelation wurde bisher keine axiomatische Theorie angegeben. Für den π-Kalkül existieren diverse Varianten der Bisimulation, die sich durch den Zeitpunkt der Substitution von Bezeichern unterscheiden. Während die *early*-Bisimulation Terme vergleicht, in denen die Substitution schon durchgeführt wurde, vergleicht die *late*-Bisimulation unsubstituierte Terme und wird daher mit Substitutionen parametrisiert. Für beide Varianten wurden axiomatische Semantiken definiert [PS95]. Für den *Fusion*-Kalkül existiert diese Unterscheidung aufgrund seiner besonderen Form von Substitution durch die Angabe von Namensäquivalenzen nicht. Die Bisimulationen für $\mathcal{P}$ entsprechen der *early*-Bisimulation des π-Kalküls, da hier die Ersetzung der Bezeichner durch gelesene Werte ebenfalls direkt in der Semantik vorgenommen wird, so daß die Bisimulationen bereits substituierte Terme in Beziehung setzen.

Da die anderen mobilen Kalküle keine Daten außer den Kanalnamen verwenden, ist die axiomatische Semantik für $\mathcal{P}$ die erste axiomatische Semantik, die die Äquivalenz von Termen, die sowohl Mobilität als auch Datenausdrücke beinhalten, beschreibt.

7.2.2 Interaktionsdiagramme

Die n-Agentendiagramme sind eine Variante von Interaktionsdiagrammen, die die Interaktion zwischen Systemkomponenten bzw. zwischen dem System und seiner Umgebung graphisch darstellen. Wir führen in diesem Abschnitt einen zusammenfassenden Vergleich mit den Interaktionsdiagrammen von UML und den Message Sequence Charts durch. Hierzu diskutieren wir verschiedene Merkmale der Diagramme.

Art der Darstellung. Die n-Agenten-Diagramme stellen wie die MSCs und die Sequenzdiagramme aus UML die Systemkomponenten als Lebenslinien dar, die den zeitlichen Verlauf der Ereignisse in einer Komponente repräsentieren. Die Nachrichten werden in Form von Pfeilen symbolisiert, wobei die verschiedenen Arten von Nachrichten durch die Art der Pfeile gekennzeichnet werden. In diesen

Diagrammen steht die Interaktion zwischen den Instanzen im Vordergrund, so daß auf die Beziehungen zwischen den Instanzen nicht eingegangen wird. Die Kollaborationsdiagramme aus UML richten ihre Fokus eher auf die Beziehungen zwischen den Instanzen. Die Nachrichten werden als Attribute der Relationen betrachtet; ihre Reihenfolge wird durch Sequenznummern spezifiziert.

Systemmodell. MSCs und UML abstrahieren von Annahmen über die Struktur der spezifizierten Systeme. Daher lassen sich diese Notationen sowohl für die Darstellung von Prozeduraufrufen in sequentiellen Systemen als auch für die Modellierung von Kommunikation in verteilten Systemen einsetzen. Allerdings können durch die eher allgemeine Natur der Sprachkonstrukte semantische Mehrdeutigkeiten entstehen (siehe unten). Daher sind die n-Agenten-Diagramme speziell für die Spezifikation verteilter Systeme entworfen, in denen eine feste Anzahl von Instanzen mit jeweils lokalem Speicher interagieren. Auf diese Weise besitzen alle Sprachkonstrukte eine eindeutige Semantik. Allerdings lassen sich prozedurale Konstrukte nicht direkt darstellen, sondern müssen simuliert werden. Zum Beispiel können Prozeduraufrufe durch Anfragen dargestellt werden.

Während MSCs nur asynchrone Kommunikation unterstützen, erlauben die n-Agenten-Diagramme die Unterscheidung zwischen synchroner und asynchroner Kommunikation. In UML wird zwischen Nachrichten unterschieden, bei denen der Sender auf den Erhalt einer Antwortnachricht warten muß, und denen, bei denen der Sender nach dem Senden der Nachricht unabhängig in seiner Ausführung fortfahren kann. Dem entspricht in den n-Agenten-Diagrammen das Konzept der Anfragen.

Die n-Agenten-Diagramme verfügen durch das Konstrukt zur dynamischen Erzeugung neuer Tornamen über die Möglichkeit, die Generierung neuer Kommunikationsverbindungen und die Vergabe entsprechender Zugriffsrechte zu spezifizieren. Diese Art von Modellierung muß in den MSCs und in UML informell erfolgen.

Generative Spezifikation. Die n-Agenten-Diagramme ermöglichen die Erstellung generativer Spezifikationen, die eine Menge von möglichen Systemabläufen beschreiben. Dies wird durch komplexe Sprachkonstrukte wie Schleifen und Komposition von Diagrammen erreicht. MSCs enthalten vergleichbare Sprachkonstrukte für Schleifen und Alternativen. Aufgrund des Systemmodells der n-Agenten-Systeme wird bei Kontrollstrukturen, die mehr als eine Instanz betreffen, eine Instanz als Kontrollinstanz benannt, die die für die Durchführung der Kontrollstruktur notwendigen Entscheidungen fällt und die anderen beteiligten Instanzen informiert. In MSCs muß die Auswahloperation nicht expliziert modelliert werden, daher entstehen an dieser Stelle Probleme wie die globale bzw. die verzögerte Auswahl [BM95, GHRW98]. Über die Möglichkeiten der n-Agenten-Diagramme zur Diagrammkomposition hinaus erlauben MSCs durch die Verwendung von

High-level MSCs die beliebige hierarchische Verschachtelung von Diagrammen.

In der Definition der Sequenzdiagramme wird auf die Verwendung von Schleifen und Alternativen nur rudimentär eingegangen. Es sind Sprachkonzepte zum wiederholten Senden von Nachrichten vorgesehen; ebenso können bedingte Nachrichten spezifiziert werden. Beide Sprachkonstrukte können in Diagrammen für verteilte Systeme zu semantischen Mehrdeutigkeiten führen [GGW98, GGW99]. Eine Zerlegung der Verhaltensbeschreibung in mehrere Diagramme wie in den n-Agenten-Diagrammen oder den HMSCs ist in UML nicht vorgesehen und muß daher zusätzlich informell angegeben werden.

Daten. Die n-Agenten-Diagramme erlauben die Integration von funktionalen Sprachen, die über ein ausreichend mächtiges Typsystem verfügen und für die die Auswertung von Ausdrücken durch eine Reduktionssemantik definiert wird. Es werden die Gültigkeitsbereiche von Bezeichnern festgelegt, insbesondere auch in Verbindung mit den komplexen Sprachkonstrukten. In MSC '96 können Daten nur informell als Nachrichtenparameter oder als Kommentar in internen Aktionen angegeben werden, so daß aus dem Diagramm allein keine Aussagen über Definiertheit und Gültigkeit von Bezeichnern getroffen werden können. In MSC 2000 ist eine Integration von Daten vorgesehen. Dabei werden keine Annahmen über die verwendete Datensprache getroffen; neben funktionalen können auch imperative Sprachen verwendet werden. Die Frage der Gültigkeitsbereiche von Bezeichnern wird in MSC 2000 nur in Form einer kurzen informellen Beschreibung behandelt. Insbesondere werden wenig Angaben über die Beziehungen zwischen Gültigkeitsbereichen und komplexen Sprachkonstrukten gemacht. MSC 2000 unterscheidet zwischen statischen und dynamischen Daten: Statische Daten werden zu Beginn eines Diagramms übergeben und sind allen Instanzen in diesem Diagramm bekannt. Sie können auch nicht während der Ausführung des in dem Diagramm beschriebenen Verhaltens geändert werden. Die dynamischen Daten werden bei der Ausführung von Diagrammen erzeugt und sind nur innerhalb der erzeugenden Instanz bekannt. Das Konzept der globalen statischen Daten widerspricht der Lokalitätsannahme bzgl. der Daten der n-Agenten-Systeme, daher ist die Parameterübergabe in unseren Diagrammen instanzbezogen. In MSCs können Nachrichten von allen Instanzen empfangen werden, die sich im Definitionsbereich dieser Nachricht befinden. Hingegen ist in den n-Agenten-Diagrammen die Zuordnung von Nachrichten an Tore der Instanzen eindeutig, so daß eine Nachricht stets einer Instanz zugeordnet ist.

UML erlaubt die Modellierung der Daten eines Systems durch diverse Diagramme wie z.B. Klassendiagramme. Mit Hilfe dieser Diagramme kann eine objektorientierte Beschreibung der Daten erzeugt werden. Allerdings fehlt in den Sequenzdiagrammen die Möglichkeit, die in den Klassendiagrammen spezifizierten Daten in die Verhaltensbeschreibung zu integrieren. So existiert kein Konzept für die Definition von Bezeichnern und ihren Gültigkeitsbereichen.

Textuelle Notation. Wie die n-Agenten-Diagramme verfügen MSCs über eine standardisierte textuelle Notation, die die textuelle Darstellung der Diagramme erlaubt. Beide Notationen sind instanzorient; die Ereignisse auf der Lebenslinie einer Instanz werden durch sequentielle Komposition ihrer textuellen Repräsentationen dargestellt. Während die Notation der n-Agenten-Diagramme nur die Instanzen der Diagramme beschreibt, enthält die Notation der MSCs auch Angaben über die statischen Daten von Diagrammen.

Für UML existiert keine standardisierte textuelle Darstellung, so daß in Arbeiten zur Definition der Semantik von UML oft eine eigene Notation eingeführt wird (z.B. in [Stö99]).

Semantik. Die Semantik von UML wird durch ein Metamodell beschrieben, in dem die Bedeutung der Sprachelemente in Form von UML-Diagrammen definiert wird [OMG99]. Diese Semantik definiert die Struktur von Diagrammen, macht aber keine Aussagen über das von den dynamischen Modellen spezifizierte Systemverhalten. Daher ist für eine formale Beschreibung der möglichen Systemabläufe die Entwicklung zusätzlicher Semantiken notwendig. Hier ergibt sich das Problem, wie die zusätzliche Semantik in die Beschreibung der Daten in Form des Metamodells von UML integriert werden kann.

Die Bedeutung von MSC '96 wird im Standard durch eine prozeßalgebraische Semantik definiert [MR94a, ITU96a]. In [MR97] wird eine Erweiterung dieser Semantik für High-level MSCs vorgestellt. Diese Semantik verwendet einen komplexen Operator für die Konkatenation von Diagrammen; für diesen Operator wird in [GHRW98] eine alternativer einfacherer Operator vorgeschlagen. Die Semantiken für MSCs definieren nur das Systemverhalten und abstrahieren von Daten. Für den neuen Standard MSC 2000, der Datenbeschreibungen in den Diagrammen beinhaltet, ist noch keine formale Semantik definiert. Daher liegt für MSCs noch keine Semantik vor, die sowohl Daten- als auch Verhaltensaspekte von Diagrammen in einer der Semantik aus Kapitel 6 vergleichbaren Weise integriert. Durch die Verwendung von Kontrollinstanzen bei Schleifen und der Diagrammkonkatenation kann in den n-Agenten-Diagrammen das Problem des *global choice* nicht auftreten, so daß diese Konstrukte eine eindeutige Semantik besitzen.

7.3 Ausblick

Wir diskutieren in diesem Abschnitt mögliche weiterführende Forschungsaktivitäten.

7.3.1 Entwicklung von Werkzeugen

Die Anwendbarkeit von Formalismen zur Spezifikation reaktiver Systeme ist in hohem Maße von der Verfügbarkeit von Werkzeugen, die die Modellierung und Analyse von Systemen unterstützen, abhängig. Zu diesen Werkzeugen gehören Programme zur Eingabe von Spezifikationen, zur Überprüfung auf syntaktische Korrektheit und korrekte Typisierung sowie Programme zur Analyse von Systemeigenschaften. Ein Beispiel für ein Analyseprogramm wäre ein Werkzeug, daß die Umformung von Termen aus $\mathcal{P}$ durch die Anwendung der Gleichungen der axiomatischen Theorie (Abbildungen 3.4 bis 3.6) ermöglicht.

Für den Kalkül $\mathcal{P}_\lambda$ wurde ein Programm entwickelt, das die Überprüfung von Spezifikationen auf lexikalische, syntaktische und semantische Korrektheit erlaubt [Bah00]. Das Programm wurde in der funktionalen Sprache *Objective Caml* [L$^+$00] implementiert. Funktionale Sprachen bieten durch die enthaltenen abstrakten Datenypen eine einfache Möglichkeit zur Darstellung der abstrakten Syntax von Spezifikationen und zur direkten Manipulation syntaktischer Strukturen, z.B. für die Berechnung von Typausdrücken. Die Entwicklung von Programmteilen für die lexikalische und die syntaktische Analyse wird von *Objective Caml* durch spezielle Generatoren, die den Werkzeugen *lex* und *yacc* [MB91] vergleichbar sind, unterstützt.

Weiterhin ist die Realisierung einer graphischen Umgebung für n-Agentendiagramme wünschenswert, die die Erstellung von Spezifikationen am Bildschirm unterstützt. Auf diese Weise können alle in der Notation enthaltenen graphischen Konstrukte zur Spezifikation von Systemen verwendet werden. Zusätzlich wäre die Integration einer semantischen Analyse in das graphische Entwicklungswerkzeug sinnvoll, die z.B. die korrekte Typisierung der Diagramme überprüft. Hierbei ist auch die Einbindung existierender UML-Werkzeuge zu untersuchen, die für die Erstellung von Spezifikationen genutzt werden können. Diese Werkzeuge müssen dann in die Analysewerkzeuge für die Diagramme eingebunden werden.

7.3.2 Fallstudien

Die Verwendung der n-Agenten-Diagramme zur integrierten Spezifikation von Verhaltens- und Datenelementen reaktiver Systeme sollte durch die Durchführung größerer, praxisbezogener Fallstudien erprobt werden. Hierzu gehört auch die Analyse von Eigenschaften der Spezifikation, z.B. durch die Anwendung der axiomatischen Semantik von $\mathcal{P}$ (siehe die Abbildungen 3.4 bis 3.6). Die Fallstudien sollten unterschiedliche Anforderungen an die Datenbeschreibung stellen, damit die Verwendung verschiedener Datensprachen untersucht werden kann. Mögliche Fallbeispiele sind die Fertigungszelle aus dem KorSo-Projekt [LL94c, LL94b, Mua98] sowie das Bankenbeispiel aus [Inf96].

Zur Durchführung umfangreicherer Fallstudien ist die Unterstützung durch geeignete Werkzeuge wünschenswert, um die Eingabe von Spezifikationen und

die Überprüfung auf ihre statische Korrektheit möglichst einfach durchführen zu können. Insbesondere zur Anwendung der axiomatischen Semantik von $\mathcal{P}$ zur Analyse von Systemen ist die Werkzeugunterstützung notwendig, um die schrittweise Umformung der Terme halbautomatisch durchführen zu können.

7.3.3 Kalküle mit Datenbehandlung

In diesem Abschnitt beschreiben wir mögliche Erweiterungen des in dieser Arbeit vorgestellten Kalküls. Weiterhin diskutieren wir die Erweiterung verwandter mobiler Kalküle um Daten.

Integration weiterer Datensprachen

Der $\mathcal{P}$-Kalkül aus Kapitel 3 erlaubt die Integration verschiedener Datensprachen. In Kapitel 4 wird als Beispiel für eine solche Integration ein erweiterter, getypter λ-Kalkül vorgestellt. Der so entstandene Kalkül $\mathcal{P}_\lambda$ erlaubt die problemorientierte Beschreibung von Daten durch seine Datentypen, die Möglichkeit zur Rekursion und Funktionen höherer Ordnung. Allerdings fehlen in der Datensprache Konzepte, die aus den funktionalen Programmiersprachen bekannt sind [Rea89, Thi94]. Hierzu gehören z.B. die Definition von Funktionen mittels mehrerer Regeln, die sich durch den erwarteten Inhalt der Funktionsargumente unterscheiden (*pattern matching*) sowie die Behandlung von Ausnahmen (*exception handling*). Daher sollte zum einen die Datensprache aus Kapitel 4 um die fehlenden Konzepte erweitert werden, zum anderen sollten Fallstudien mit weiteren Datensprachen durchgeführt werden. Durch die Definition unterschiedlicher Sprachkonzepte in verschiedenen Datensprachen kann so ein "Werkzeugkasten" entstehen, aus dem je nach Spezifikationsaufgabe die geeignete Datensprache in den Prozeßkalkül integriert werden kann. Auf diese Weise kann auch der Komplexitätsgrad der Datensprache und damit auch die Komplexität der zugehörigen Reduktionssemantik an das zu spezifizierende Problem angepaßt werden.

Behandlung von Ausnahmen

Während die Einführung rein funktionaler Konzepte wie z.B. des *pattern matching* nur eine Veränderung innerhalb der Datensprache bedeutet, wäre bei der Realisierung der Ausnahmenbehandlung die Definition eines neuen Operators der Prozeßteilsprache sinnvoll, um auch in der Verhaltensspezifikation das Reagieren auf das Auftreten von Ausnahmen modellieren zu können. Ein erster Ansatz hierfür ist der *test*-Operator aus *ProFun* [Geh96], der die Behandlung von Ausnahmen auf Prozeßebene erlaubt. Neben der Definition der operationellen Semantik eines solchen Operators wäre auch die Ergänzung der axiomatischen Semantik in den Abbildungen 3.4 und 3.5 wünschenswert, um auch für Terme mit Ausnahmebehandlung Äquivalenzbetrachtungen durchführen zu können.

Fusion Calculus

Wie bereits in Abschnitt 3.6 auf 52 angeführt, ist der Fusion Calculus [PV98] von Parrow und Victor eine Variante des asynchronen π-Kalküls, bei der die Unterscheidung zwischen Eingabeaktionen, die Bindungen erzeugen, und Ausgabeaktionen aufgehoben ist. Anstelle der Ersetzung empfangener Werte im Restterm des Empfängers werden *fusions* verwendet; hierbei handelt es sich um spezielle Transitionsbeschriftungen, die Äquivalenzklassen über Kanalnamen definieren.

Ein möglicher Aspekt weiterer Forschungsarbeit ist die Integration von Datenspezifikation in den Fusion Calculus. Hierzu müssen die *fusions* modifiziert werden, um anstelle von Äquivalenzen auf Kanalnamen nun die Bindung von Werten an Bezeichner darzustellen. Die *fusions* realisieren dann die aus Programmiersprachensemantiken bekannten Umgebungen. Dabei ist die Gültigkeit von Bezeichnern zu beachten, da im Fusion Calculus die Äquivalenzen in einer *fusion* immer für das Gesamtsystem gültig sind, wenn dies nicht explizit durch Bindung von Namen ausgeschlossen wird. Daher müssen die Gültigkeitsbereiche von Bezeichnern explizit definiert werden. Ebenso ist zu beachten, daß im Fusion Calculus Eingabe- und Ausgabeaktionen symmetrisch sind; daher ist zu untersuchen, wie die Bindung gelesener Werte unter Beibehaltung dieser Symmetrie realisiert werden kann.

Mobile Ambients

Der *Ambients*-Kalkül [CG97] von Cardelli und Gordon ist ein Prozeßkalkül, der neben Prozessen und Kommunikation auch Lokalitäten beinhaltet. Eine Lokalität, genannt *ambient*, ist ein definierter Ort, an dem parallele Prozesse ablaufen können. Lokalitäten werden durch Namen identifiziert und können hierarchisch verschachtelt werden. Durch *Migration* können untergeordnete *ambients* zu anderen *ambients* verschoben werden bzw. aus *ambients* ausgelagert werden. Durch seine Fähigkeit zur Modellierung von Lokalitäten ist der *Ambients*-Kalkül besonders für die Spezifikation von Sicherheitseigenschaften und die Verwaltung von Zugriffsrechten geeignet [Car00].

Analog zur Verwendung von Kanalnamen im π-Kalkül verwendet der *Ambients*-Kalkül die Namen der Lokalitäten als einzigen "Datentyp". Um eine vollständige und problemorientierte Spezifikation von reaktiven Systemen aus dem Sicherheitsbereich zu unterstützen, wäre eine Erweiterung des *Ambients*-Kalküls um Datenbehandlung sinnvoll. Hierbei ist insbesondere die Natur der Lokalitäten zu diskutieren. Sind Lokalitäten wie die Kanäle in $\mathcal{P}$ Werte eines speziellen Datentyps, ist auch die Verwendung von Lokalitäten in Datenausdrücken möglich. Dies kann die Definition einer komplexen Semantik und Äquivalenztheorie zur Folge haben, da die Sprache dann über eine große Mächtigkeit verfügt (z.B. können Listen von Lokalitäten verwaltet werden). Werden hingegen die Lokalitäten als eigenes, von der Datensprache getrenntes Konzept aufgefaßt, kann dies die Defini-

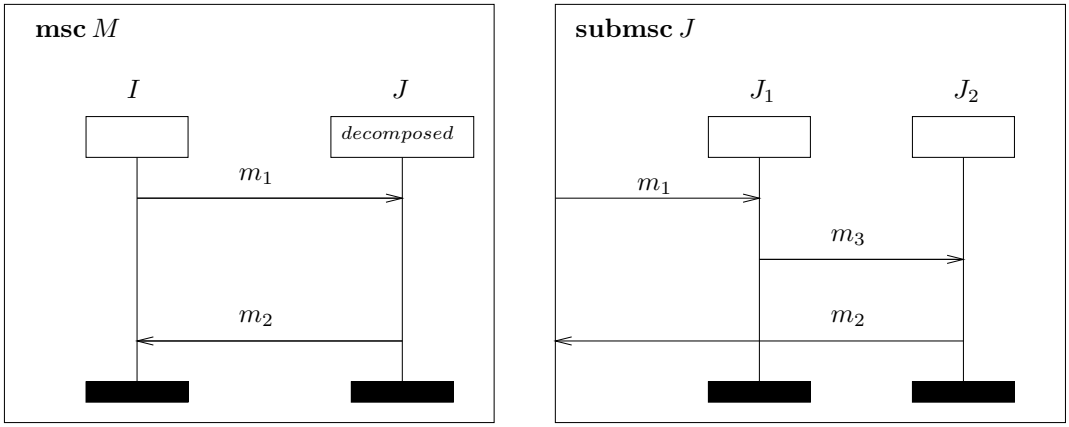


Abbildung 7.2: Instanzdekomposition in Message Sequence Charts.

tion sowohl der operationellen als auch der axiomatischen Semantik vereinfachen; hingegen verliert die Sprache gegenüber der ersten Variante an Ausdrucksfähigkeit.

7.3.4 Graphische Notationen

Nun diskutieren wir mögliche Ansätze für weitere Forschungsarbeiten auf dem Gebiet der graphischen Notationen mit Daten.

Instanzdekomposition

Für Message Sequence Charts ist das Verfahren der *Instanzdekomposition* vorgesehen [ITU96a]. Bei diesem Verfahren wird eine einzelne Instanz durch ein Diagramm ersetzt, das das interne Verhalten als Folge von Nachrichten zwischen den lokalen Komponenten der Instanz beschreibt.

In Abbildung 7.2 ist ein MSC M angegeben, in dem eine Instanz I eine Nachricht m_1 an eine Instanz J sendet und von dort anschließend eine Nachricht m_2 empfängt. Die Verwendung des Schlüsselworts *decomposed* zeigt an, daß für eine genauere Beschreibung des Verhaltens von J ein spezielles Diagramm existiert. Dieses Diagramm wird in Abbildung 7.2 mit **submsc** J gekennzeichnet und enthält zwei Instanzen J_1 und J_2, die die internen Komponenten von J repräsentieren. An Diagramme, die eine Instanz verfeinern, wird die Anforderung gestellt, daß dieses Diagramm alle Interaktionen, die die verfeinerte Instanz durchführt, ebenfalls durchführen muß. Daher treten im Diagramm für J die Nachrichten m_1 und m_2 auf, die als Interaktion mit der Umgebung von J dargestellt werden. Zusätzlich sendet J_1 eine lokale Nachricht m_3 an J_2, die außerhalb von J nicht sichtbar ist. Weiterhin wird m_1 von J_1 empfangen, während m_2 von J_2 gesendet wird.

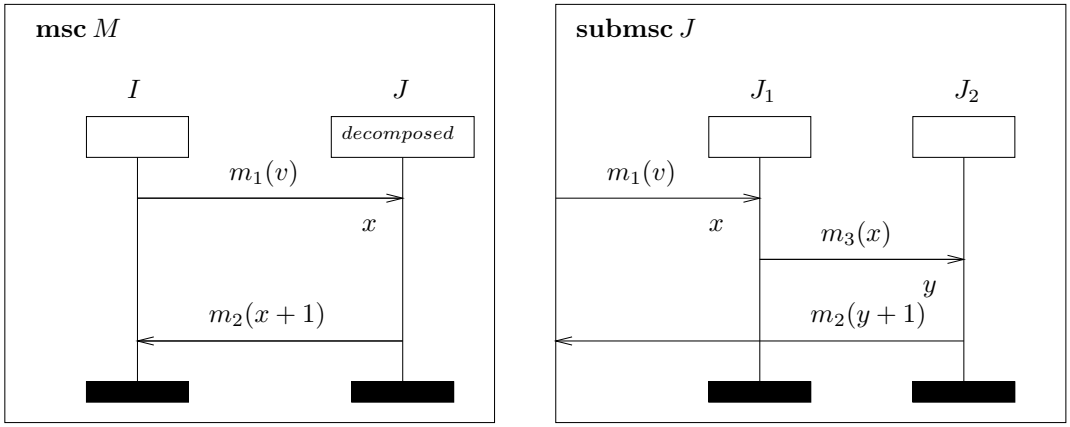


Abbildung 7.3: Instanzdekomposition mit Daten.

Bei der Integration von Daten kommen zusätzliche Anforderungen an die Instanzdekomposition hinzu. In Abbildung 7.3 wird das Beispiel aus Abbildung 7.2 um Daten erweitert. Die Nachricht m_1 überträgt einen Wert v an J. Dieser wird in J an x gebunden. Mit der Nachricht m_2 wird dann der Wert $x+1$ an I gesendet. Im Diagramm mit der Dekomposition von J wird v von J_1 empfangen und dort an den Bezeichner x gebunden. Dieser Bezeichner ist nur in J_1 gültig. Da die Nachricht m_2 von J_2 gesendet wird, kann diese Nachricht nicht den Parameter $x+1$ enthalten. Im Diagramm für J wird daher zunächst der Wert von x an J_2 übertragen, dort an y gebunden und anschließend mittels m_2 der Wert $y+1$ an I gesendet. Das Diagramm J realisiert somit eine dem Verhalten von J äquivalente Ausführung.

Die Semantik für MSC '96 [ITU96b] beinhaltet die Beschreibung der Instanzdekomposition, berücksichtig aber generell nicht die Daten in Diagrammen, da Daten in MSC '96 nur informell verwendet werden. Die Beschreibung der Instanzdekomposition für MSC 2000 in [ITU99] geht auf die Einhaltung der Datenkonsistenz bei der Verfeinerung von Instanzen nicht ein. Eine formale Semantik für MSC 2000 existiert noch nicht.

Bei der Integration der Instanzdekomposition in die n-Agenten-Diagramme sind Bedingungen festzulegen, unter denen die Verfeinerung einer Instanz und die Instanz im übergeordneten Diagramm äquivalentes Verhalten zeigen. Hierbei ist insbesondere die Verwendung von Daten zu berücksichtigen. Weiterhin muß die operationelle Semantik um die Beschreibung der Instanzdekomposition erweitert werden.

Hierarchische Komposition von Diagrammen

In n-Agentendiagrammen ist nur die sequentielle Konkatenation von Diagrammen vorgesehen, wobei die Angabe von alternativen Fortsetzungen möglich ist.

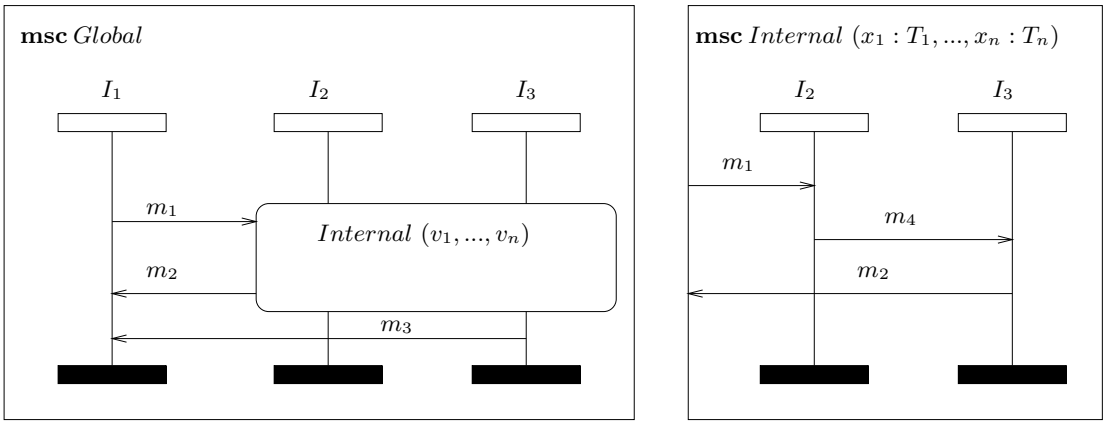


Abbildung 7.4: Verschachtelung von Diagrammen.

Im Standard MSC 2000 [ITU99] ist darüberhinaus die Verschachtelung von Diagrammen erlaubt.

In Abbildung 7.4 ist ein MSC-Dokument angegeben, das die beiden Diagramme *Global* und *Internal* beinhaltet. Das Diagramm *Internal* wird in *Global* als untergeordnetes Diagramm verwendet, das einen Abschnitt des Verhaltens der beiden Instanzen I_2 und I_3 beschreibt. Beim Aufruf des Diagramms wird ein Vektor von Parametern übergeben; diese Parameter werden in *Internal* in den Variablen x_i gespeichert, deren Inhalt in *Internal* nicht geändert werden darf. Während der Ausführung von *Internal* interagieren die Instanzen I_2 und I_3 mit I_1, nach Beendigung von *Internal* sendet I_3 eine Nachricht m_3 an I_1.

Die Verwendung hierarchischer Komposition von Diagrammen unterstützt den schrittweisen Entwurf reaktiver Systeme. Zudem kann auf diese Weise das Verhalten einer Teilmenge der Instanzen beschrieben werden. Aus diesem Grund ist die Integration eines vergleichbaren Konzepts in die n-Agenten-Diagramme wünschenswert. Um die Wiederverwertbarkeit von Spezifikationen zu erhöhen, wäre es sinnvoll, auch die Instanznamen als Parameter an untergeordnete Diagramme übergeben zu können. Auf diese Weise ließen sich bestimmte standardisierte Abläufe modellieren (z.B. Kommunikationsprotokolle), die dann in neue Spezifikationen integriert und von den dort verwendeten Instanzen ausgeführt werden können.

Optionales Verhalten

In [DH99] wird von Damm und Harel eine, *Live Sequence Charts* (LSC) genannte, Variante der *Message Sequence Charts* vorgestellt. LSCs erlauben die Unterscheidung zwischen obligatorischen (*mandatory*) und optionalem (*provisional*) Verhalten. Auf diese Weise lassen sich, besonders in den frühen Phasen der

Modellierung, Entwürfe für mögliche Teile von Systemausführungen als optional beschreiben, die anschließend in den weiteren Phasen der Modellierung gegebenenfalls als obligatorisch oder als unerwünscht gekennzeichnet werden. So läßt sich eine schrittweise Annäherung an das gewünschte Systemverhalten erzielen.

Die einzelnen Diagramme innerhalb eines LSC-Dokuments können mit Aktivierungsbedingungen (*activation conditions*) versehen werden. Erfüllt ein System eine Aktivierungsbedingung, muß das Systemverhalten dem im zugehörigen Diagramm beschriebenen Verhalten entsprechen. Hierbei können Bedingungen als obligatorisch oder als optional gekennzeichnet werden. Ist eine obligatorische Bedingung zum Zeitpunkt ihrer Auswertung falsch, bricht die Systemausführung ab, da ein erwünschtes Systemverhalten nicht mehr möglich ist. Ist hingegen eine optionale Bedingung falsch, wird das im zugehörigen Diagramm beschriebene Verhalten nicht ausgeführt, sondern das Systemverhalten wird, z.B. mit dem Testen anderer Bedingungen, fortgesetzt. Weiterhin kann angegeben werden, ob eine Nachricht in jedem möglichen Systemablauf übertragen werden soll oder ob sie nur in einigen der möglichen Abläufe übertragen wird. Ebenso kann angegeben werden, ob das auf einer Lebenslinie einer Instanz angegebene Verhalten optional oder obligatorisch ist.

Bei einer Erweiterung der n-Agenten-Diagramme um die Unterscheidung zwischen optionalem und obligatorischem Verhalten müssen Mechanismen entwickelt werden, wie die Spezifikation von optionalem Verhalten im Zusammenhang mit Daten verwendet werden kann. Enthält z.B. ein Ausdruck in einer obligatorischen Datendefinition einen Bezeichner, der innerhalb optionalen Verhaltens definiert wird, muß der Ausdruck auch dann reduziert werden können, wenn das optionale Verhalten nicht ausgeführt wird. Eine Möglichkeit der Realisierung wäre ein Konzept zur Belegung von Bezeichnern mit *default*-Werten. Ein anderer, restriktiverer Ansatz bestände in der Festlegung von Regeln bzgl. des Datenflusses, die definieren, daß Berechnungen in obligatorischen Abschnitten oder Nachrichten nur Bezeichner aus anderen obligatorischen Abschnitten verwenden dürfen.

Weiterhin muß untersucht werden, in welcher Form die Aktivierungsbedingungen angegeben werden können. In dem Systemmodell der n-Agenten-Systeme, die aus verteilten Instanzen bestehen, ist die Verwendung von Aktivierungsbedingungen sinnvoll, die, analog zu den Bedingungen bei *while*-Schleifen und bei Konkatenation, jeweils einer Kontrollinstanz zugeordnet werden, die über die Gültigkeit der Bedingung entscheidet.

Darüber hinaus ist die Entwicklung einer geeigneten Methodik sinnvoll, die die schrittweise Konkretisierung des Systemverhaltens mittels der Angabe von optionalem bzw. obligatorischem Verhalten unterstützt.

Echtzeit

Bei der Spezifikation von reaktiven Systemen gewinnt die Modellierung von Echtzeiteigenschaften der Systeme zunehmend an Bedeutung, da Systeme z.B. in tech-

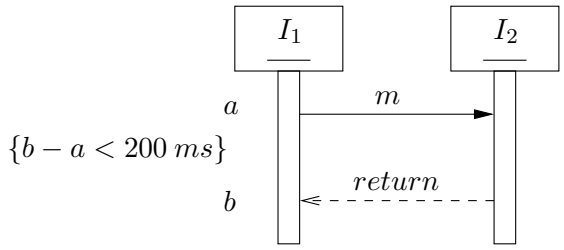


Abbildung 7.5: Sequenzdiagramm mit Echtzeit.

nischen Steuerungen eingebaut werden, in denen die Einhaltung von Echtzeitanforderungen essentiell ist.

Ein Beispiel für eine Echtzeitanforderung ist die Angabe maximaler Antwortzeiten von Anfragen. In Abbildung 7.5 ist ein Sequenzdiagramm aus UML mit Echtzeit [Dou97] angegeben, das eine Anfrage beschreibt. Der Zeitpunkt des Sendens der Anfrage wird mit a gekenntzeichnet, der Zeitpunkt des Empfangens der Antwort mit b. In einer Bedingung (*constraint*) wird festgelegt, daß die Antwort auf die Anfrage in weniger als 200 Millisekunden beim Sender eingetroffen sein muß. Jeder Ablauf, in dem diese Bedingung nicht eingehalten wird, ist ein ungültiger Systemablauf.

Eine Möglichkeit zur Verifikation der Einhaltung von Echtzeitbedingungen ist die Verwendung von *model checking* [CGP99]. Hierbei wird das Sequenzdiagramm mit Echtzeit in ein Modell übersetzt, daß die Verifikation von Zeitbedingungen unterstützt. Eine erste Untersuchung zur Analyse eines solchen Systems wird in [FHD+99] durchgeführt. Hierbei werden die Anforderungen an ein HiFi-Protokoll mit Sequenzdiagrammen mit Echtzeitbedingungen modelliert. Die Diagramme werden dann in *timed automata* [AD94], eine Klasse von zeitbehafteten Automaten, übersetzt. Anschließend wird gezeigt, daß eine gegebene *timed automata*-Spezifikation des Protokolls die in den Diagrammen festgelegten Echtzeitanforderungen erfüllt. Hierzu wird das Programmpaket UPPAAL [LPY97] verwendet.

Für eine Erweiterung der n-Agenten-Diagramme um die Behandlung von Echtzeit ist zu diskutieren, in welcher Form Zeitaspekte in das Systemmodell aufgenommen werden sollen. Zum einen könnte Zeit als eigenes Konzept in die Diagramme aufgenommen werden, zum anderen ließe sich Zeit in Form eines speziellen Datentyps als Teil der Datensprache betrachten. Im ersten Fall wäre die Semantik der Zeitaspekte einfacher, da sie sich direkt in die Semantik der Verhaltensteilsprache integrieren ließe. Im zweiten Fall wäre die Ausdrucksfähigkeit größer, da Zeitangaben wie die Werte anderer Datentypen in Ausdrücken verwendet werden könnten. Dies würde aber eine komplexere Semantik implizieren.

Weiterhin wäre zu untersuchen, wie die bekannten Techniken zur Analyse zeitbehafteter Spezifikationen an die Behandlung von Spezifikationen mit Daten

anzupassen sind.

Literaturverzeichnis

[AD94] Rajeev Alur, David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–225, 1994.

[AG96] Ken Arnold, James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[AHP96] Rajeev Alur, Gerard J. Holzmann, Doron Peled. An Analyzer for Message Sequence Charts. In Tiziana Margaria, Bernhard Steffen (Hrsg.), *Proceedings of TACAS '96*, Band 1055 von *Lecture Notes in Computer Science*, S. 35–48. Springer, 1996.

[ALT95] R. Amadio, L. Leth, B. Thomsen. From a concurrent lambda-calculus to the pi-calculus. In *Proc. of Foundations of Computation Theory*, Band 965 von *Lecture Notes in Computer Science*, 1995.

[Ama94] Roberto M. Amadio. Translating Core Facile. Technischer Bericht ECRC–94–3, European Computer–Industry Research Centre GmbH, Arabellastraße 17, 81925 München, 1994.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21:613–641, 1978.

[Bah00] Florian Bahr. Implementierung einer lexikalischen, syntaktischen und semantischen Analyse für ein parametrisierbares, mobiles Kalkül. Technische Universität Braunschweig, Institut für Software, Abteilung Programmierung, 2000.

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North–Holland, 1984.

[Bar90] H.P. Barendregt. Functional Programming and Lambda Calculus. In *[vL90]*, S. 321–363. 1990.

[BAS97] H. Ben-Abdallah, S.Leue. Syntactic Detection of Process Divergence and non-Local Choice in Message Sequence Charts. In E. Brinksma

(Hrsg.), *Proceedings of TACAS '97*, Band 1217 von *Lecture Notes in Computer Science*, S. 259–274. Springer, 1997.

[BB87] Tommaso Bolognesi, Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BHR84] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, Juli 1984.

[BIM95] Bard Bloom, Sorin Istrail, Albert R. Meyer. Bisimulation Can't Be Traced. *Journal of the ACM*, 42(1):232–268, Januar 1995.

[BM95] J.C.M. Baeten, S. Mauw. Delayed choice: an operator for joining Message Sequence Charts. In D. Hogrefe, S. Leue (Hrsg.), *Formal Description Techniques VII*, S. 340–354. Chapman & Hall, 1995.

[BMS96] Manfred Broy, Stephan Merz, Katharina Spies (Hrsg.). *Formal System Specification: The RPC-Memory Specification*, Band 1169 von *Lecture Notes in Computer Science*. Springer, 1996.

[BRJ98] Grady Booch, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[BS00] F.L. Bauer, R. Steinbrüggen (Hrsg.). *Foundations of Secure Computation*, Band 175 von *NATO Science Series F: Computer and System Sciences*. IOS Press, 2000.

[BV92] Jos C. M. Baeten, Frits W. Vaandrager. An Algebra for Process Creation. *Acta Informatica*, 29(4):303–334, 1992. Report version: CS-R8907, CWI, Amsterdam; also available in "J.W. de Bakker, 25 Jaar Semantiek — Liber Amicorum", Stichting Mathematisch Centrum, Amsterdam, 1989.

[BV93] J.C.M. Baeten, C. Verhoef. A congruence theorem for structured operational semantics with predicates. In E. Best (Hrsg.), *Proceedings of CONCUR '93*, Band 715 von *Lecture Notes in Computer Science*, S. 477–492. Springer, 1993.

[BW90] J.C.M. Baeten, W.P. Weijland. *Process Algebra*, Band 18 von *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[Car00] Luca Cardelli. Mobility and Security. In [BS00], 2000.

[CG97] Luca Cardelli, Andrew D. Gordon. Mobile Ambients. Technischer Bericht, University of Cambridge, Juli 1997.

[CGP99] Edmund M. Clarke, Orna Grumberg, Doron Peled. *Model Checking*. MIT Press, 1999.

[DeM79] Tom DeMarco. *Structured Analysis and Systems Specification*. Prentice Hall, 1979.

[DH99] Werner Damm, David Harel. LSC's: Breathing Life into Message Sequence Charts. In *Proceedings of FMOODS '99*, 1999.

[Dou97] Bruce Powel Douglass. *Real-Time UML – Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1997.

[End97] Tom Ender. *Object-Oriented Programming with REXX*. John Wiley and Sons, Inc., 1997.

[Ern92] Marcel Erné. *Einführung in die Ordnungstheorie*. B.I.-Wissenschaftsverlag, 1992.

[FH95] W. Ferreira, M. Hennessy. Towards a Semantic Theory of CML. Technischer Bericht 2/95, University of Sussex, Februar 1995.

[FHD+99] Thomas Firley, Michaela Huhn, Karsten Diethers, Thomas Gehrke, Ursula Goltz. Timed Sequence Diagrams and Tool-Based Analysis - A Case Study. In Robert France, Bernhard Rumpe (Hrsg.), *Proceedings of the Second International Conference on the Unified Modeling Language (<< UML >> '99)*, Band 1723 von *Lecture Notes in Computer Science*, S. 645–660. Springer, 1999.

[FHJ95] William Ferreira, Matthew Hennessy, Alan Jeffrey. A Theory of Weak Bisimulation for Core CML. Technischer Bericht 05/95, University of Sussex, September 1995.

[Fir97] Thomas Firley. Erweiterung eines Compiler für ProFun. Studienarbeit, Institut für Informatik, Universität Hildesheim, Mai 1997.

[Fir98] Thomas Firley. Spezifikation und Verifikation eines RPC-Speichers in der Sprache ProFun. Diplomarbeit, Februar 1998.

[Fok00] Wan Fokkink. *Introduction to Process Algebra*. Springer, 2000.

[Geh96] Thomas Gehrke. Eine Programmiersprache für verteilte Systeme mit funktionalem Datenanteil. Diplomarbeit, Institut für Informatik, Universität Hildesheim, 1996.

[Geh98] Thomas Gehrke. An Algebraic Semantics for an Abstract Language with Intra-Object-Concurrency. In David Pritchard, Jeff Reeve (Hrsg.), *Proceedings of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par '98)*, Band 1470 von *Lecture Notes in Computer Science*, S. 733–737. Springer, 1998. Erweiterte Version als Hildesheimer Informatik-Bericht HIB 7/98, Universität Hildesheim, Institut für Informatik, Mai 1998.

[Geh00] Thomas Gehrke. Interaktionsdiagramme mit Datenspezifikation zur Darstellung verteilter Systeme. In *Tagungsband der GI-Jahrestagung Informatik 2000*. Springer, September 2000.

[GF99] Thomas Gehrke, Thomas Firley. Generative Sequence Diagrams with Textual Annotations. In Katharina Spies, Bernhard Schätz (Hrsg.), *Proceedings of the 9th GI/ITG-Workshop "Formale Beschreibungstechniken für verteilte Systeme" (FBT '99)*, S. 65–72. Herbert Utz Verlag, 1999.

[GGW98] Thomas Gehrke, Ursula Goltz, Heike Wehrheim. The Dynamic Models of UML: Towards a Semantics and its Application in the Development Process. Hildesheimer Informatik-Bericht 11/98, Universität Hildesheim, Institut für Informatik, September 1998.

[GGW99] Thomas Gehrke, Ursula Goltz, Heike Wehrheim. Zur semantischen Analyse der dynamischen Modelle von UML mit Petri-Netzen. In E. Schnieder (Hrsg.), *Tagungsband der 6. Fachtagung "Entwicklung und Betrieb komplexer Automatisierungssysteme"*, S. 547–566. Braunschweig, 1999.

[GH96] Thomas Gehrke, Michaela Huhn. ProFun – a Language for Executable Specifications. In H. Kuchen, S.D. Swierstra (Hrsg.), *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP '96)*, Band 1140 von *Lecture Notes in Computer Science*, S. 304–318. Springer, 1996. Erweitere Version als Hildesheimer Informatik-Bericht 17/96, Universität Hildesheim, Institut für Informatik, Juni 1996.

[GHRW98] Thomas Gehrke, Michaela Huhn, Arend Rensink, Heike Wehrheim. An Algebraic Semantics for Message Sequence Chart Documents. In Stan Budkowski, Ana Cavalli, Elie Najm (Hrsg.), *Proceedings of FORTE/PSTV '98*, S. 3–18. Kluwer Academic Press, 1998. Erweiterte Version als Hildesheimer Informatik-Bericht HIB 5/98, Universität Hildesheim, Institut für Informatik, Mai 1998.

[GMP89] A. Giacalone, P. Mishra, S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2), 1989.

[Gol88] Ursula Goltz. *Über die Darstellung von CCS–Programmen durch Petrinetze*. GMD–Bericht Nr. 172. R. Oldenbourg, 1988.

[GP94a] J. F. Groote, A. Ponse. Proof Theory for μCRL: A Language for Processes with Data. In D. J. Andrews, J. F. Groote, C. A. Middelburg (Hrsg.), *Proceedings of the International Workshop on Semantics of Specification Languages (SOSL'93)*, Workshops in Computing, S. 232–251, London, UK, Oktober 1994. Springer.

[GP94b] J. F. Groote, A. Ponse. The Syntax and Semantics of μCRL. In A. Ponse, C. Verhoef, S. F. M. van Vlijmen (Hrsg.), *Algebra of Communicating Processes, Utrecht 1994*, S. 26–62. Workshops in Computing, Springer-Verlag, 1994.

[GR97] Thomas Gehrke, Arend Rensink. Process Creation and Full Sequential Composition in a Name-Passing Calculus. Hildesheimer Informatik-Bericht HIB 7/97, Institut für Informatik, Universität Hildesheim, Mai 1997.

[GR99] Thomas Gehrke, Arend Rensink. A Mobile Calculus with Data. Technischer Bericht 99-04, Technische Universtität Braunschweig, Institut für Software, Abteilung Programmierung, Oktober 1999. Available at `http://www.cs.tu-bs.de/ips/gehrke/publications.html`.

[GRG93] Peter Graubmann, Ekkart Rudolph, Jens Grabowski. Towards a Petri Net Based Semantics Definition for Message Sequence Charts. In *Proceedings of the 6th SDL Forum (SDL '93)*, 1993.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[Har97] David Harel. Vorwort zu [Dou97], 1997.

[Hav94] Klaus Havelund. *The Fork Calculus*. PhD thesis, DIKU, University of Kopenhagen, 1994.

[HI93] M. Hennessy, A. Ingólfsdóttir. A Theory of Communicationg Processes with Value Passing. *Information and Computation*, 107:202–236, 1993.

[HL94] Klaus Havelund, Kim G. Larsen. The Fork-Calculus. *Nordic Journal of Computing*, 1:346–363, 1994.

[HMM86] Robert Harper, David MacQueen, Robin Milner. Standard ML. Technischer Bericht ECS–LFCS–86–2, Department of Computer Science, University of Edinburgh, 1986.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hof88] Douglas R. Hofstadter. *Metamagicum – Fragen nach der Essenz von Geist und Struktur*. Klett–Cotta, 1988.

[Hog89] Dieter Hogrefe. *Estelle, LOTOS und SDL – Standard-Spezifikationssprachen für verteilte Systeme*. Springer, 1989.

[HP98] David Harel, Michal Politi. *Modeling Reactive Systems with Statecharts*. McGraw-Hill, 1998.

[Inf96] Banken & Konten – Ein Beispiel zur Spezifikation nebenläufiger Informationssysteme. Arbeitspapier des Workshops "Concurrency in Information Systems", Technische Universität Braunschweig, Institut für Software, Abteilung Informationssysteme, Januar 1996.

[ISO87] ISO – Information Processing Systems – Open Systems Interconnection. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. ISO 8807, 1987.

[ISO97] ISO/IEC JTC1/SC2/WG7, Project WI 1.21.20.2.3. *Working Draft on Enhancements to LOTOS*, Januar 1997.

[ITU96a] International Telecommunication Union ITU. Message Sequence Chart (MSC). Standard ITU-T Z.120, 1996.

[ITU96b] International Telecommunication Union ITU. Message Sequence Chart (MSC). Standard ITU-T Z.120 (Annex B), 1996.

[ITU99] International Telecommunication Union ITU. MSC 2000. Standard ITU-T Z.120, November 1999.

[JCJÖ92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard. *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley, 1992.

[Jef96] Alan Jeffrey. Semantics for core Concurrent ML using computation types. Computer Science Report 3/96, University of Sussex, 1996.

[JF92] J.F.Groote, F.W.Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, 1992.

[JR91] Kurt Jensen, Grzegorz Rozenberg. *High-level Petri-nets. Theory and Application*. Springer, 1991.

[JW85] Kathleen Jensen, Niklaus Wirth. *Pascal User Manual and Report – ISO Pascal Standard*. Springer, 1985.

[Kna99] Alexander Knapp. A Formal Semantics for UML Interactions. In Robert France, Bernhard Rumpe (Hrsg.), *Proceedings of the Second International Conference on the Unified Modeling Language (<< UML >> '99)*, Band 1723 von *Lecture Notes in Computer Science*, S. 116–130. Springer, 1999.

[L+00] Xavier Leroy u.a. *The Objective Caml system – release 3.0*, April 2000. Erhältlich unter `ftp://ftp.inria.fr/lang/caml-light/ocaml-3.00-refman.pdf`.

[LG91] Rita Loogen, Ursula Goltz. Modelling Nondeterministic Concurrent Processes with Event Structures. *Fundamenta Informaticae*, 14:39–74, 1991.

[LL94a] P.B. Ladkin, S. Leue. What Do Message Sequence Charts Mean? In M.U. Uyar (Hrsg.), *Proceedings of the 6th International Conference on Formal Description Techniques*, S. 301–316. North-Holland, 1994.

[LL94b] Claus Lewerentz, Thomas Lindner. Case Study Production Cell: A Comparative Study in Formal Software Development. FZI-Publication 1/94, Forschungszentrum Karlsruhe, 1994.

[LL94c] Claus Lewerentz, Thomas Lindner. Formal Methods for Reactive Systems – The Comparative Case Study "Production Cell". Technischer Bericht, Forschungszentrum Karlsruhe, 1994. Availiable at `ftp.fzi.de/pub/korso/production_cell/tutorial`.

[LPY97] Kim G. Larsen, Paul Pettersson, Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1+2), 1997.

[Man88a] José A. Manas. A Tutorial on ADT semantics for LOTOS users – Part I: Fundamental Concepts. Technischer Bericht, Dpt. Ingeniería Telemática, E.T.S.I. Telecomunicación, Ciudad Universitaria Madrid, 1988.

[Man88b] José A. Manas. A Tutorial on ADT semantics for LOTOS users – Part II: Operations on Types. Technischer Bericht, Dpt. Ingeniería Telemática, E.T.S.I. Telecomunicación, Ciudad Universitaria Madrid, 1988.

[MB91] Tony Mason, Doug Brown. *lex & yacc*. O'Reilly & Associates Inc., 1991.

[Mic96] Christian Michel. Getting Started with Object REXX. In *Proceedings of the SHARE Technical Conference*, März 1996.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil90] Robin Milner. Functions as Processes. Technischer Bericht 1154, INRIA, Februar 1990.

[Mil93] Robin Milner. The polyadic π-calculus: a tutorial. In Bauer, Brawer, Schwichtenberg (Hrsg.), *Logic and Algebra of Specification*. Springer, 1993.

[MP92] Zohar Manna, Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer, 1992.

[MPW92] Robin Milner, Joachim Parrow, David Walker. A Calculus of Mobile Processes, Part I+II. *Information and Computation*, 100, 1992.

[MR94a] S. Mauw, M.A. Reniers. An Algebraic Semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.

[MR94b] S. Mauw, M.A. Reniers. An algebraic semantics of Message Sequence Charts. Computing Science Notes 94-23, Department of Computing Science, Eindhoven University of Technology, April 1994.

[MR97] S. Mauw, M.A. Reniers. High-level Message Sequence Charts. In *Proceedings of SDL '97:Time for Testing – SDL, MSC and Trends*. Elsevier, 1997.

[MS92] Robin Milner, Davide Sangiorgi. Barbed Bisimulation. In *Proceedings of ICALP '92*, Band 623 von *Lecture Notes in Computer Science*, S. 685–695. Springer, 1992.

[MTH90] Robin Milner, Mads Tofte, Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Mua98] Rami Muallem. Implementierung einer Fallstudie in ProFun, Evaluierung der Sprache und ihre Übersetzung nach Java. Diplomarbeit, Universität Hildesheim, Institut für Informatik, April 1998.

[NN96] Flemming Nielson, Hanne Riis Nielson. From CML to its process algebra. *Theoretical Computer Science*, (155):179–219, 1996.

[OFMP+94] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, J.R.W. Smith. *Systems Engineering Using SDL-92*. North Holland, 1994.

[OMG99] Object Management Group OMG. Unified Modeling Language Specification. Technischer Bericht, Version 1.3 R9, 1999.

[P+96] John Peterson u.a. Report on the Programming Language Haskell – Version 1.4. Technischer Bericht, University of Glasgow, 1996.

[Pae98] Barbara Paech. On the Role of Activity Diagrams in UML. In Pierre-Alain Muller, Jean Bézivin (Hrsg.), *Proceedings of << UML >> '98*. University of Haute-Alsace, Mulhouse, France, 1998.

[Pau91] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[Plo81] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technischer Bericht DAIMI FN–19, Computer Science Department, University of Århus, 1981.

[Pnu86] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, Band 224 von *Lecture Notes in Computer Science*, S. 510–584. Springer, 1986.

[Pra86] Vaughan Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[PS95] Joachim Parrow, Davide Sangiorgi. Algebraic theories for name-passing calculi. *Information and Computation*, 120:174–197, 1995.

[PT95] Benjamin C. Pierce, David N. Turner. Pict Language Definition. Technischer Bericht, University of Cambridge, 1995.

[PT97] Benjamin C. Pierce, David N. Turner. Pict: A Programming Language Based on the Pi-Calculus. CSCI Technical Report 476, Indiana University, 1997.

[PV98] Joachim Parrow, Björn Victor. The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes (extended abstract). In *Proceedings of LICS '98*. IEEE Press, 1998.

[Rea89] Chris Reade. *Elements of Functional Programming*. Addison–Wesley, 1989.

[Rei82] Wolfgang Reisig. *Petri Nets – An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer, 1982.

[Rei90] Wolfgang Reisig. *Petrinetze – Eine Einführung*. Studienreihe Informatik. Springer, 1990.

[Ren92] Arend Rensink. Posets for Configurations! In *Proceedings of CONCUR '92*, Band 630 von *Lecture Notes in Computer Science*. Springer, 1992.

[Ren99] M.A. Reniers. *Message Sequence Chart – Syntax and Semantics*. PhD thesis, Technische Universiteit Eindhoven, 1999.

[Rep89] J.H. Reppy. First-class synchronous operations in Standard ML. Technischer Bericht, Cornell University, 1989.

[Rep92] J.H. Reppy. Concurrent ML: Design, application and semantics. In *Programming, Concurrency, Simulation and Automated Reasoning*, Band 693 von *Lecture Notes in Computer Science*, S. 165–198. Springer, 1992.

[Rep99] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[RJB98] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[RS99] Christine Röckl, Davide Sangiorgi. A π-calculus Process Semantics of Concurrent Idealised ALGOL. In Wolfgang Thomas (Hrsg.), *Proceedings of FOSSACS '99*, Band 1578 von *Lecture Notes in Computer Science*, S. 306–321. Springer, 1999.

[RW94] Arend Rensink, Heike Wehrheim. Weak Sequential Composition in Process Algebras. In B. Jonsson, J. Parrow (Hrsg.), *Proceedings of CONCUR '94*, Band 836 von *Lecture Notes in Computer Science*, S. 226–241. Springer, 1994.

[Sch88] Alois Schütte. *Programmieren in Occam*. Addison–Wesley, 1988.

[Sch92] Uwe Schöning. *Theoretische Informatik kurz gefasst*. BI-Wissenschaftsverlag, 1992.

[Shu88] Ken Shumate. *Understanding Concurrency in Ada*. McGraw–Hill, 1988.

[Ste88] Ralf Steinmetz. *Occam 2*. Hüthig, 1988.

[Stö99] Harald Störrle. A Petri-net semantics for Sequence Diagrams. In Katharina Spiess, Bernhard Schätz (Hrsg.), *Tagungsband des 9. GI/ITG-Fachgesprächs "Formale Beschreibungstechniken für verteilte Systeme" (FBT '99)*, S. 233–242. Herbert Utz Verlag, 1999.

[Str92] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison–Wesley, 1992.

[TDT95] S. Tucker Taft, R. A. Duff, T. Taft (Hrsg.). *Ada95 Reference Manual: Language and Standard Libraries: Intenational Standard ISO/IEC 8652:1995(E)*, Band 1246 von *Lecture Notes in Computer Science*. Springer, 1995.

[TG97] J. Thees, R. Gotzhein. A Formal Syntax and a Formal Semantics for Open Estelle. Technischer Bericht 292/97, Insitut für Informatik, Universität Kaiserslautern, 1997.

[Thi94] Peter Thiemann. *Grundlagen der funktionalen Programmierung*. Teubner, 1994.

[Tho90] Wolfgang Thomas. Automata on Infinite Objects. In *[vL90]*, S. 133–191. 1990.

[Tho95] Bent Thomsen. A Theory of Higher Order Communicating Systems. *Information and Computation*, 116:38–57, 1995.

[TLG92] Bent Thomsen, Lone Leth, Alessandro Giacalone. Some Issues in the Semantics of Facile Distributed Programming. In *Proceedings of REX Workshop "Semantics: Foundations and Applications"*, Band 666 von *Lecture Notes in Computer Science*. Springer, 1992.

[TLP+93] Bent Thomsen, Lone Leth, Sanjiva Prasad, Tsung-Min Kuo, Andre Kramer, Fritz Knabe, Alessandro Giacalone. Facile Antigua Release Programming Guide. Technischer Bericht ECRC–93–20, European Computer–Industry Research Centre GmbH, Arabellastrasse 17, 81925 München, 1993.

[Tur85] B.A. Turner. Miranda. A nonstrict functional language with polymorphic types. In *Functional Programming Languages and Computer Architectures*, Band 201 von *Lecture Notes in Computer Science*. Springer, 1985.

[vG90] R.J. van Glabbeek. The Linear Time – Branching Time Spectrum. In J.C.M. Baeten, J.W. Klop (Hrsg.), *Proceedings of CONCUR '90*, Band 458 von *Lecture Notes in Computer Science*, S. 278–297. Springer, 1990.

[vG93] R.J. van Glabbeek. The Linear Time – Branching Time Spectrum II. In Eike Best (Hrsg.), *Proceedings of CONCUR '93*, Band 715 von *Lecture Notes in Computer Science*, S. 66–82. Springer, 1993.

[vL90] J. van Leeuwen (Hrsg.). *Handbook of Theoretical Computer Science – Volume B: Formal Models and Semantics*. Elsevier Science Publishers, 1990.

[Wal95] David Walker. Objects in the π-Calculus. *Information and Computation*, 116:253–271, 1995.

[Wat97] Christoph Watermann. Darstellung der Mobilitätsaspekte im π-Kalkül durch die Spezifikationssprache LOTOS. Diplomarbeit, Institut für Informatik, Universität Hildesheim, Oktober 1997.

[Win87] Glynn Winskel. Event structures. In *Advances in Petri Nets*, Band 255 von *Lecture Notes in Computer Science*. Springer, 1987.

Index

Anhang A

Glossar

$\mathcal{B}$	Basiskalkül	Abschnitt 2.1
$\mathcal{P}$	Prozeßkalkül	Abschnitt 3.2
$\mathcal{P}_\lambda$	Prozeßkalkül mit Datensprache	Abschnitt 4.1
c	Kanalwert	Abschnitt 2.1
C	Menge von Kanalwerten	Abschnitt 2.1
E	Ausdruck	Abschnitt 3.1
t	Prozeßterm	Abschnitt 2.1
v	Wert der Datensprache	Abschnitt 3.1
x	Bezeichner	Abschnitt 3.1
$\vec{\cdot}$	Vektor	Abschnitt 3.1
CExpr	Geschlossene Ausdrücke	Definition 4.1
Chan	Menge der Kanalwerte	Abschnitt 3.1
Chan^T	Kanäle, die Werte des Typs T übertragen	Abschnitt 3.1
Diag	Menge der Diagramme	Definition 6.2
DType	Typsystem der Datensprache	Abschnitt 4.1
Expr	Menge der Ausdrücke	Abschnitt 4.1
Expr^T	Ausdrücke des Typs T	Abschnitt 3.1
Gates	Menge der Tore	Definition 6.2
GType	Typsystem der n-Agenten-Diagramme	Definition 6.1
Inst	Menge derInstanzen	Definition 6.2
Type	Typsystem des Prozeßkalküls	Definition 3.2
Value	Menge der Werte der Datensprache	Abschnitt 4.1
Value^T	Menge der Werte des Typs T	Abschnitt 3.1
Var	Menge der Bezeichner	Abschnitt 3.1

$\mathbf{0}$	Deadlock	Abschnitt 2.1
$\mathbf{1}$	Erfolgreich terminierter Prozeß	Abschnitt 2.1
τ	Interne Aktion	Abschnitt 2.1
$E!F$	Ausgabeaktion	Abschnitt 3.2
$E?x;t$	Eingabeaktion	Abschnitt 3.2
$\{\vec{x} \leftarrow \vec{E}\};t$	Definition von Bezeichnern	Abschnitt 3.2
$[E]$	Test-Operator	Abschnitt 3.2
$spawn(t)$	Prozeßerzeugung	Abschnitt 2.1
$t + t$	Auswahloperator	Abschnitt 2.1
$t;t$	Sequentielle Komposition	Abschnitt 2.1
$\mathbf{ch}\ C.\,t$	Kanalrestriktion	Abschnitt 2.1
$P(\vec{E})$	Prozeßaufruf	Abschnitt 3.2
$c_{\$T\$}$	Kanalwert des Typs T *channel*	Abschnitt 3.1
Δ	Umgebung für Typannahmen	Abschnitt 3.3
Θ	Prozeßumgebung	Abschnitt 3.2
Ω	Menge der Transitionsbeschriftungen	Abschnitt 3.4
$\xrightarrow{\omega}$	Transitionsrelation	Definition 3.4
$\stackrel{\omega}{\Longrightarrow}$	Schwache Transitionsrelation	Abschnitt 3.5.3
$\hookrightarrow$	Reduktionsrelation der Datensprache	Definition 4.1
$=_\alpha$	α-Äquivalenz	Definition 3.3
$=_\beta$	β-Äquivalenz	Definition 4.4
$\sim_\beta$	Starke Bisimulation	Definition 3.7
$\approx_\beta$	Schwache Bisimulation	Definition 3.12
$\simeq_\beta$	Schwache Kongruenz	Definition 3.15
$\mathcal{AX}_{\sim_\beta}$	Axiom. Theorie für starke Bisimulation	Abschnitt 3.5.2
$\mathcal{AX}_{\simeq_\beta}$	Axiom. Theorie für schwache Kongruenz	Abschnitt 3.5.3
$fv(.)$	Menge der freien Bezeichner	Abschnitt 3.1
$fc(.)$	Menge der nicht-restringierten Kanäle	Abschnitt 3.1
$\langle \vec{E}/\vec{x} \rangle$	Substitution	Abschnitt 3.4
$\ll d/c \gg$	α-Konversion	Definition 3.3

$[\![.]\!]$	Reduktionssemantik eines Ausdrucks	Definition 4.1
$L(x)$	Vom Nichtterminal x erzeugte Sprache	Abschnitt 5.4.1
$[\![.]\!]^{\mathcal{E}}_{expr}$	Übersetzung von Ausdrücken	Abbildung 6.1
$[\![.]\!]_{document}$	Übersetzung von Dokumenten	Abbildung 6.2
$[\![.]\!]^{D}_{diagram}$	Übersetzung eines Diagramms	Abbildung 6.2
$[\![.]\!]^{D}_{instance}$	Übersetzung einer Instanz	Abbildung 6.2
$[\![.]\!]^{D,I,\mathcal{E}}_{event}$	Übersetzung eines Ereignisses	Abbildung 6.3
$[\![.]\!]^{D,I,\mathcal{E}}_{comm\_event}$	Übersetzung von Kommunikation	Abbildung 6.5
$[\![.]\!]^{D,I,\mathcal{E}}_{data}$	Übersetzung von Datendefinitionen	Abbildung 6.7
$[\![.]\!]^{D,I,\mathcal{E}}_{gate}$	Übersetzung von Torgenerierung	Abbildung 6.7
$[\![.]\!]^{D,I,\mathcal{E}}_{gate}$	Übersetzung von Alternativen	Abbildung 6.8
$[\![.]\!]^{D,I,\mathcal{E}}_{loop}$	Übersetzung von *loop*-Schleifen	Abbildung 6.9
$[\![.]\!]^{D,I,\mathcal{E}}_{while}$	Übersetzung von *while*-Schleifen	Abbildung 6.10
$[\![.]\!]^{D,I,\mathcal{E}}_{continue}$	Übersetzung von Konkatenation	Abbildung 6.11

D	Diagrammname	Abschnitt 6.3
I	Instanzname	Abschnitt 6.3
m	Torname	Abschnitt 5.2
$\mathcal{E}$	Menge der bekannten Bezeichner	Abschnitt 6.3
$bound\_by(.)$	Gebundene Bezeichner	Abbildung 6.4
$chan(m, I)$	Kanal für Tor m in Instanz I	Abschnitt 6.1
$gate(c)$	Inverse Funktion zu $chan(m, I)$	Abschnitt 6.1
$gates(I)$	Tore der Instanz I	Abschnitt 6.1
$init()$	Initiales Diagramm eines Dokuments	Definition 6.7
$initpar(D, I)$	Initiale Parameter von I in Diagramm D	Definition 6.8
$make\_vect(.)$	Sortieren einer Bezeichnermenge in Vektor	Definition 6.6
$name(.)$	Torname ohne Typangabe: $name(c_{\$T\$}) = c$	Abschnitt 6.1
$to\_set(.)$	Inverse Funktion zu *make_vect*.	Definition 6.6
$type\_conv(.)$	Umwandlung GType → DType	Abschnitt 6.1
$type\_of(m, I)$	Typ des Tornamens m in Instanz I	Abschnitt 6.1